

U  
S  
E  
U NIX  
USENIX

WORKSHOP  
PROCEEDINGS

MACH

October 4-5, 1990  
Burlington, Vermont

For additional copies of these proceedings contact

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710 USA

The price is \$17 for members and \$20 for nonmembers.

Outside the U.S.A and Canada, please add  
\$9 per copy for postage (via air printed matter).

Copyright © 1990 by The USENIX Association  
All rights reserved.

This volume is published as a collective work.  
Rights to individual papers remain  
with the author or the author's employer.

UNIX is a registered trademark of AT&T.  
Other trademarks are noted in the text.



# Program and Table of Contents

## Mach Workshop October 4-5, 1990

### Thursday, October 4

<i>Opening Session</i>	9:00 - 10:15	
Introduction		
Melinda Shore, mt Xinu		
Keynote Speech		
Dr. Richard F. Rashid, Carnegie-Mellon University		
<i>Break</i>	10:15 - 10:45	
<i>Memory Management</i>	10:45 - 12:15	
Zone Garbage Collection		1
James Van Sciver, Open Software Foundation, and Richard F. Rashid, Carnegie-Mellon University		
Extending the Mach External Pager Interface to Accomodate User-Level Page Replacement Policies		17
Dylan McNamee and Katherine Armstrong, University of Washington		
Mach on a Virtually Addressed Cache Architecture		31
Chia Chao, Milon Mackey, and Bart Sears, Hewlett-Packard Laboratories		
<i>Lunch</i>	12:15 - 1:30	

<i>Internals and Performance</i>	1:30 - 3:30	
The Mach Timing Facility: An Implementation of Accurate Low-Overhead Usage Timing David L. Black, Carnegie-Mellon University		53
Real-Time Mach: Towards a Predictable Real-Time System Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao, Carnegie-Mellon University		73
Developing Benchmarks to Measure the Performance of the Mach Operating System David Finkel et al., Worcester Polytechnic Institute		83
A Revised IPC Interface Richard Draves, Carnegie-Mellon University		101
<i>Break</i>	3:30 - 4:00	
<i>Threads Panel Discussion</i>	4:00 - 5:00	
<b>Friday, October 5</b>		
Open Software Foundation Papers To Be Announced	9:00 - 10:30	
<i>Break</i>	10:30 - 11:00	
<i>Environments and Applications</i> A Persistent Distributed Architecture Supported by the Mach Operating System Francis Vaughan et al., University of Adelaide	11:00 - 12:30	123
An Ultrix 4.0 Uniserver Daniel E. Geer, DEC Cambridge Research Laboratory		
A Trusted X Window System Server for Trusted Mach Jeremy Epstein and Marvin Shugerman, TRW Systems Division		141

<i>Lunch</i>	12:30 - 2:00	
<i>Fault Tolerance</i>	2:00 - 3:30	
Building a Fault-Tolerant System Based on Mach		157
Rong Chen and Tony P. Ng, University of Illinois at Urbana-Champaign		
Transparent Recovery of Mach Applications		169
Arthur Goldberg et al., IBM TJ Watson Research Center		
Fault-Tolerant Computing Based on Mach		185
Özalp Babaoglu, University of Bologna		
<i>Break</i>	3:30 - 4:00	
<i>Works in Progress</i>	4:00 - 5:00	

#### Program Committee

Melinda Shore, mt Xinu, Chair  
 Alan Langerman, Encore Computer Corporation  
 Douglas Orr, Chorus systemes  
 Homayoon Tajalli, Trusted Information Systems  
 Avadis Tevanian, NeXT, Inc.



# Zone Garbage Collection

James Van Sciver  
Open Software Foundation  
Richard F. Rashid  
Carnegie-Mellon University

October, 1990

## 1. Introduction

Zones are a kernel memory allocator originally implemented at CMU as part of the Mach kernel. They provide a fast allocation/deallocation mechanism for fixed size kernel objects. This is done by allocating a pool of wired down kernel memory, breaking this pool up into fixed size elements, populating a zone with those elements, and satisfying requests for specific objects from that object's zone. The use of the zones subsystem can be found throughout the Mach kernel.

As originally implemented, the amount of memory allocated to a zone could increase but not decrease. Over time, heavy zone usage would decrease the number of free kernel pages. This problem was corrected at CMU for the Mach micro-kernel by adding garbage collection to the zones package.

OSF/1, the Open Software Foundation's operating system offering, is based on Mach 2.5. OSF adopted the CMU zone garbage collection work for inclusion in OSF/1 and, in the process, "commercialized" the work. This paper describes the zones code, the zones garbage collector added by CMU, and the process OSF used to adopt CMU's technology.

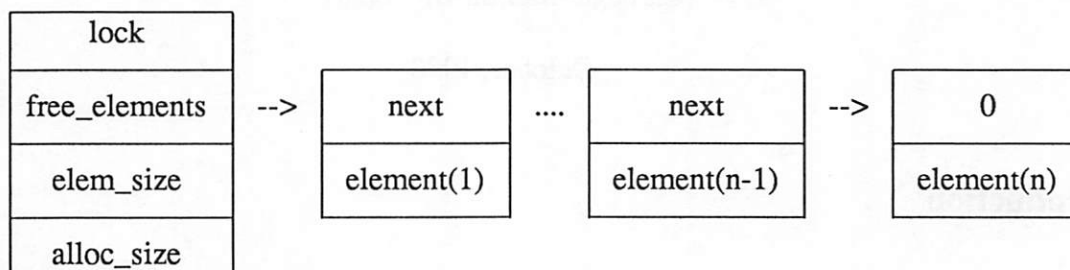
## 2. Introduction to the Zone Memory Allocator

This first section is a detailed description of the zone memory allocator. It is provided as background for the subsequent discussion of the zones garbage collector. Zones usage is pervasive in the Mach kernel but this is not an externally visible system. As a result, there has been a dearth of documentation on their structure and behavior. This section attempts to correct that lack of information. The reader who is familiar with Mach internals and the zone package may want to skim or skip this section.

When reading one should realize that the word "allocate" is used liberally to describe two operations. The first being the allocation of memory from the general kernel memory pool to a zone. The second being the allocation of an element from a zone. The word "free" describes the converse operations: returning an element to a zone and returning zone memory to the general pool.

## 2.1 Zone Structure

The essence of a zone structure is relatively simple. The pieces of the structure actively used by the zone allocate and free routines are a pointer to a null terminated list of free elements, a simple lock, the element size in bytes, and an allocation size.



**Figure 1.** Overview of zone structure

The simple lock is used to control access to any field in the zone structure. Simple locks can be optionally compiled out of uniprocessor versions of Mach so, this field is not always extant. The lock is primarily acquired by the zone allocate and free routines prior to modifying the free element list.

The free\_elements pointer is the head of a null terminated, singly linked list of available elements. The zones allocate and free routines use the first bytes of a free element to store the free list pointer, the implication being that an element is at least as large as a pointer.

The zone element's size is used when newly allocated memory is being added into a zone and, as we will see, in the garbage collection code. Remember that the element size must be at least the size of a pointer. In fact, the zones initialization code will always round a given element's size up to the nearest longword boundary to both meet this condition and to take advantage of the possible performance advantage of longword aligned data.

Alloc\_size is the number of bytes to allocate from kernel memory whenever the zone requires more elements. The zone initialization code will always round alloc\_size up to the number of bytes in a Mach page. This field is only used by zalloc, the zone allocator.

In practice, the zone structure is a little more complicated than our original example. (See the next figure.) The additional complexity primarily provides debugging information and does not contribute to the zone's operation.

```

typedef struct zone {
    decl_simple_lock_data(lock) /* generic lock */
    int count; /* Number of elements used now */
    vm_offset_t free_elements;
    vm_size_t cur_size; /* current memory utilization */
    vm_size_t max_size; /* how large can this zone grow */
    vm_size_t elem_size; /* size of an element */
    vm_size_t alloc_size; /* size used for more memory */
    boolean_t doing_alloc; /* is zone expanding now */
    char *zone_name; /* a name for the zone */
    unsigned int
    /* boolean_t */ pageable :1, /* zone pageable */
    /* boolean_t */ exhaustible :1, /* merely return if empty */
    /* boolean_t */ collectable :1; /* garbage collect empty pages */
    lock_data_t complex_lock; /* Lock for pageable zones */
    struct zone * next_zone; /* Link for all-zones list */
} *zone_t;

```

**Figure 2.** Zone Structure, from zalloc.h

Count can be very useful for debugging and statistics gathering but is not actually used by the zalloc and zfree routines. Count is the current number of allocated zone elements. As such, it is decremented when elements are removed, and incremented when they have been returned. This measure can be useful when looking for memory leaks.

The cur\_size and max\_size fields are also useful debugging aids. They are used to put a cap on a zone's growth. The behavior that occurs when cur\_size exceeds max\_size depends on the exhaustible flag: if a zone is marked as exhaustible, the allocation routines will simply return zero, if the exhaustible flag is not set then a system panic occurs.

The complex lock and the two flags, pageable and doing alloc, are used for pageable zones. The pageable feature of the zones package supports zones filled with unwired memory, but no subsystems currently use a pageable zone and pageable zones are not collectable, so they will not be addressed here.

The zone\_name is essential to zone debugging but does not contribute to a zone's operation. The next\_zone pointer used to fit into the category of useful but unused until its inclusion in the garbage collection scheme. It is now used by the collection routine to walk through the system's list of zones.

## 2.2 Global variables

In addition to having a lightweight structure, the zones code uses only a few global variables. The zone\_zone variable is the pointer to the "zone of zones". This is the pool of available zone structures. It is interesting that the zone subsystem uses itself to allocate zone structures for



newly created zones. This wrinkle allows zones to grow as easily as any of the subsystems which use it.

This scheme creates a small chicken and egg problem at system startup time. The zone package must be initialized before the general memory system is available since the latter allocates its objects, such as vm objects and vm maps, from zones. This problem is solved by using a small amount of fixed memory allocated early in the memory initialization to both create and populate the zone of zones. The external variables `zdata` and `zdata_size` are the address and size of this initial memory.

Once a booting system has created `zone_zone` and initialized its memory, `zone_init` is called to initialize the `zone_map`, a submap of the kernel map. All subsequent zones are filled with memory from this submap. There are two reasons why the zone subsystem allocates its memory from `zone_map`. The first is to avoid possible deadlock with the kernel map. Zone use pervades the kernel and many of its callers will be holding the kernel map's lock. The second reason is the more benevolent. Using a submap will decrease the number of entries to search on behalf of both the kernel and the zones, yielding a corresponding increase in performance.

The next set of global zone variables define the list of all zones. This list was previously only used for debugging but is now also used by the garbage collection routine. This set consists of:

- `first_zone` - the head of the singly linked list
- `last_zone` - a pointer to the last zone in the list
- `num_zones` - current number of zones
- `all_zones_lock` - a simple lock.

The `all_zones_lock` protects the previous three variables and the `next_zone` field in the zone structure. It is the top level in the two level zone locking hierarchy. The bottom level is the lock that protects an individual zone.

There is also a set of variables that are used by the internal zone routine `zget_space`. These are `zget_space_lock` and the static variables this lock protects: `zalloc_next_space` and `zalloc_end_of_space`. The `zget_space` routine is a contiguous memory allocator that was intended to decrease fragmentation. Zone garbage collection precludes the use of `zget_space`, so it will only be discussed in terms of relative memory efficiency.

The last global variable is the `zone_ignore_overflow` boolean. When this flag is set the use of all zones' maximum fields is negated and zones can grow without bound.

## 2.3 External Interfaces

There are a small number of externally exported zone routines that are used to create a zone and access its elements. Each of these routines, with the exception of the initialization routine, are

small enough and used often enough to also warrant an in line macro form. The routines and their alternative macro forms are displayed in Figure 3.

<pre>zone_t zinit(size, max, alloc, name)     vm_size_t size; /*in*/     vm_size_t max; /*in*/     vm_size_t alloc; /*in*/     char      *name; /*in*/</pre>	
<pre>vm_offset_t zalloc(zone)     zone_t    zone; /*in*/</pre>	<pre>ZALLOC(zone, ret, type)     zone_t    zone; /*in*/     vm_offset_t ret; /*out*/</pre>
<pre>vm_offset_t zget(zone)     zone_t    zone; /*in*/</pre>	<pre>ZGET(zone, ret, type)     zone_t    zone; /*in*/     vm_offset_t ret; /*out*/</pre>
<pre>void zfree(zone, elem)     zone_t    zone; /*in*/     vm_offset_t elem; /*in*/</pre>	<pre>ZFREE(zone, element)     zone_t    zone; /*in*/     vm_offset_t elem; /*in*/</pre>

Figure 3. Zone Routines and Macros

The zinit routine will initialize a new zone and return a pointer to the new zone structure. The zone will use elements of size bytes. Alloc bytes (rounded up to a multiple of the element size) will be allocated when the zone needs to be replenished. If overflow checking is enabled the zone size is limited to max bytes.

Elements are obtained by invoking either zalloc or zget for a particular zone. Zget differs from zalloc in that it will return zero should it be necessary to restock the zone. In the case of an empty zone, zalloc would refill it and possibly block in the process. The non-blocking zget form is used when the caller, such as an interrupt service routine, could not tolerate being suspended.

An element is returned to a zone via zfree. Unlike many user library alloc and free routines, there is no "arena checking" to verify that the element being returned had originated from the given zone. This lack of double checking is reasonable given that the zones code is intended to be a very fast allocator, and that the Mach kernel handles its objects very cleanly. For example, thread structures are only zalloc'd by the thread\_create routine and zfree'd by thread\_deallocate.

The macros perform the same function as their corresponding routines. Ideally these macros would have the same calling conventions as the routine forms but language restrictions make this impossible. The ZALLOC and ZGET macros return the allocated element in the `ret` parameter. The `type` parameter is used by the macro to appropriately cast the return address as can be seen by the following excerpt:

```
(ret) = (type) (zone)->free_elements;
```

### 3. Motivation for Garbage Collection

Note that the description of how zones work concentrates on how they can expand when necessary. Unfortunately, the zone package had no mechanism to decrease the size of a zone when its elements were no longer needed. This feature would not be necessary if a small number of subsystems used the zones code and if their use were somehow limited. Unfortunately, as can be seen in the following table, this is not the case.

Zone Name	cur_size	Zone Name	cur_size
device pager structures	4080	non-kernel map entries	49016
external page bitmaps	4092	objects	48960
fictitious pages	4088	pathbufs	16384
kalloc.16	4096	pmap	4096
kalloc.32	4096	port translations	69420
kalloc.64	8192	ports	15360
kalloc.128	16384	ports (reserved)	105600
kalloc.256	36864	pv_list	12276
kalloc.512	32768	select queues	4092
kalloc.1024	8192	sets	6072
kalloc.2048	8192	small existence maps	2160
kernel map entries	45056	superblocks	40960
large existence maps	8192	tasks	19564
large messages	8212	threads	28560
mach_net messages	16384	u-areas	63248
maps	12096	ucrd	12000
messages	8192	vm_info zone	49056
mounts	18800	vnode pager structures	32256
nfsmount	4032	vnode_pager port hash	4096
nfsreq	4080	zones	16240

**Figure 4.** Active Zones and Their Current Size

This table shows all the active zones and their current sizes as measured on a DECstation 3100. The machine was an OSF development machine in its usual state: a single user running X windows with a moderate number of mounted NFS file systems. The machine was quiescent at the time the measurement was made and yet approximately 835 Kbytes of system memory

resides in the zones. Not all zones are displayed in this table. Many zones for inactive subsystems have no allocated elements.

Other than the boundaries of system memory there are no limits on the number of some objects that can be created. A rogue task could create the maximum number of threads possible. Upon exiting, the `thread_deallocate` would return the thread structures back to the threads zone but the zone would never return its pages of now free elements back to the general memory pool. As a result, free system memory gradually declines during a system's lifetime. The zone garbage collection routines developed at CMU correct this degradation.

## 4. The Mechanics of Zone Garbage Collection

Zone garbage collection can be thought of as the inverse of the `zalloc` operation. `Zalloc` will, when a zone is empty, allocate one or more pages, break up those pages into that zone's element size chunks, and link those elements onto the zones free list. The collection code does the opposite. It goes through a zones free list, looking for pages of free elements, and freeing those pages back to the kernel memory pool. The difference between the two lies in how they are invoked. The addition of elements is stimulated by a request for an element from a specific zone. The return of those elements back to general kernel memory is stimulated by a low number of free system pages. The removal of pages from a zone could occur when individual elements are freed back to a zone but, in the Mach spirit of lazy evaluation, it makes more sense to delay the collection of free zone pages until they are needed.

### 4.1 Garbage Collection Structures

The garbage collection code required minimal change to existing structures and only one additional structure. A collectable flag was the only addition to the zone structure. Its addition was primarily intended to turn collection off on a per-zone basis during development.

The `zone_page_table_entry` structure was added to account for the number of free elements in a page. Two counts are maintained for each page, the `in_free_list` count and `alloc_count`. The `alloc_count` is how many zone elements have been allocated from a page. (Note that the page could contain elements that span page boundaries. The count of elements per page includes partial elements so one element may be counted in two pages.) `In_free_list` is a count of how many zone elements are currently free. If `in_free_list` is equal to `alloc_count` then the page is eligible for garbage collection.



```

struct zone_page_table_entry {
    short in_free_list;
    short alloc_count;
};

```

**Figure 5. Zone Page Table Entry**

A small number of global variables were added to maintain the state of the garbage collection map. These variables are:

- `zone_page_table` - pointer to the `zone_page_table` array.
- `zone_map_min_address` - minimum address of the zone submap.
- `zone_map_max_address` - maximum address of the zone submap.
- `zone_page_table_lock` - locks access to the zone page table.

The zone page table array has a `zone_page_table_entry` for each page in the zone submap. The submap addresses span the range between the min and max variables. The lock controls access to the zone page table array. This lock is orthogonal to the previously mentioned zones locking hierarchy.

## 4.2 Garbage Collection Routines

The zones garbage collection implementation was very clean. Only small, isolated, internal changes were made to the existing zones routines and macros. No changes were made to the external interfaces. A number of garbage collection routines were added. All of the changes were surrounded by an `#ifdef COLLECT_ZONE_GARBAGE` definition.

### 4.2.1 Garbage Collection Support Routines

The following routines were added to support garbage collection:

- `zone_page_free` - increments `in_free_list`
- `zone_page_in_use` - decrements `in_free_list`
- `zone_page_alloc` - increments `alloc_count`
- `zone_page_dealloc` - decrements `alloc_count`
- `zone_page_init` - initializes `in_free_list` and `alloc_count`

The first five of these routines are very similar in operation. They take an address and size as arguments, verify that the given address range lies within the zone map, lock the zone page table, modify the desired count for all of the page table entries within the given range, unlock the page table, and return. These routines have no return value. The range check is necessary because some zones, such as the zone of zones, may be populated with memory that does not

originate from the zones submap. Attempts to free this memory back to the kernel pool would be disastrous. If any page in the given range is not from zone\_map then no action is taken.

These support routines are used by both the garbage collection routine, zone\_gc, and by the pre-existing zones code. Zone\_page\_init is used to initialize the in\_free\_list and alloc\_count for newly allocated memory. Zone\_page\_alloc is used to initialize the page table entries for pages being added to a zone. The zone\_page\_free and zone\_page\_in\_use routines are used whenever an element is removed or returned from a zone. They can be found in both forms of the zone allocate and free routines.

#### 4.2.2 Zone\_gc

The zone garbage collection routine, zone\_gc, will walk through all free elements in all collectable zones looking for reclaimable pages. This routine is invoked by the pageout daemon when the system free page count is low. The collection routine has two loops: the first loop handles finding collectible pages, the second loop frees those pages back to kernel memory.

The outer loop is fairly mundane. It locks the all\_zones\_lock so that it can walk through the list of zones. It then locks a zone's lock so it can walk through that zone's list of free elements.

The inner collection loop is truly elegant. The problem to be solved in zone garbage collection is that, although elements are ordered by ascending address when they are originally inserted in the free list, they are in random order at collection time. The problem is how to walk through the loop once, removing only those elements which are eligible for collection and remembering which page is to be freed.

The solution is simplified by use of the zone\_page\_table\_entry. A page is eligible for collection if its in\_free\_list and alloc\_counts are equal. All elements belonging to that page should be removed. The zone\_page macro will convert an element address into the corresponding page table entry address. Upon encountering an element that belongs to a collectible page, the page table entry's two counts are decremented and the element is removed from the zone. When the count hits zero, all elements belonging to the page have been removed from the free list and the page can be returned.

The second loop looks for page entries with zero counts, frees those pages, and changes the count to the symbol ZONE\_PAGE\_UNUSED, which has a value of negative one.

### 5. OSF's Contribution

The garbage collection work was originally done at CMU for use in the microkernel. The development group at OSF wanted to merge this work into this November's OSF/1 offering. Our operating system is based on Mach 2.5, a pre-microkernel version of Mach.

```

zone_gc()
{
    lock(all_zones_lock);
    for (all zones) {
        lock(zone);
        for (all elements in the zone's free list) {
            if (element is from the zone submap &&
                zone_page(element)->in_free_list ==
                zone_page(element)->alloc_count) {

                zone_page_in_use (element, zone->elem_size);
                zone_page_dealloc(element, zone->elem_size);

                zone->cur_size -= zone->elem_size;
                unlink this element from the free list;
            }
        }
        unlock(zone);
    }
    unlock(all_zones_lock);

    for (all pages in the zone map) {
        if (this page can be freed) {
            kmem_free(zone_map, page, PAGE_SIZE);
        }
    }
}

```

**Figure 6.** Psuedo Code for Zone Garbage Collection

The garbage collection code that OSF received from CMU provided a solid foundation from which to work. It was in working condition and all of the collection changes were clearly demarcated. The steps we took to adopt these changes were to:

1. write zone monitoring utilities and tests
2. merge the CMU changes with our Mach 2.5 source and test the merged code
3. examine and attempt to improve performance
4. deploy and document the changes

## 5.1 Zone Monitoring and Testing

The original emphasis on the monitors and tests was on robustness rather than performance. A way was needed to interactively monitor zone activity while stressing the system. The goal was to write something small, quick, and flexible. Towards these ends a data extraction utility, `zone_mon`, was written. `Zone_mon` uses the standard method of invoking the `nlist` library to



translate symbolic names into their kernel addresses. Only three symbols are used to walk a system's zone list: `first_zone`, `last_zone`, and `num_zones`. After picking up the addresses of these symbols `/dev/kmem` is read to extract zone structure information from a running kernel. The information is then minimally formatted and sent to standard out. It was convenient to add some `zone_mon` options to print information for a single zone or information in columnar format.

Next, a small collection of shell scripts was written to format the the data collected by `zone_mon` in various ways. The most useful of these was a script that printed out zone size information in a psuedo-graphical format. This script could then be used in a small loop to watch a zone's behavior over time. This was very helpful in detecting zone memory leaks and watching the effects of the collection code. Typical output of the size monitor for the threads zone can be seen below.

```
in use:    18816 |*****| 76%
cur_size:  24528

cur_size:  24528 |**| 7%
max_size: 344064
```

Figure 7. Thread zone sizes on a newly booted system

```
in use:    18480 |***| 18%
cur_size:  208320

cur_size:  208320 |*****| 160%
max_size: 344064
```

Figure 8. Thread zone sizes after creating and deleting 500 threads

The goal of the test scripts was to stress the zones package. The testing method created large numbers of threads in various usage patterns while simultaneously forcing the garbage collection routine, `zone_gc`, to run. The latter was done by adding a temporary system call that would simply invoke `zone_gc`. Doing `thread_creates` was useful because it would require operations on multiple zones (kernel stack space is created for each thread so map entries are allocated) without having too many system side effects.

## 5.2 Code Merge

Once the monitoring utilities and stress tests were available, the CMU and OSF code were merged and the result was tested. Only a few problems were found. The first was architecture specific. The space for the page table array was originally allocated from the kernel map. This wasn't a problem on most machines but the Pmax needed to lock the kernel map during TLB misses. The solution was to allocate the page table array from the zone map. There was a small

off by one problem with initialization of the zone `in_free_list` count that was only discovered by running the tests for a very long period of time.

Finally, there was a small accounting problem caused by a mismatch in the use of the `cur_size` fields between the two sets of source. OSF's source set a zone's current size to the total size of allocated memory. If elements are not integral divisors of the page size, and they rarely are, some number of bytes at the end of a page are not actually used in the zone. The garbage collection code will decrease `cur_size` by an element size when it removes all of a collectable page's elements from the zones. Therefore, as each page is allocated to and collected from a zone, `cur_size` grows by the residual number of bytes. It wasn't too long before it would exceed the `max_size` field and cause a system crash.

### 5.3 Performance Improvements

Three changes were made to the garbage collection code in an attempt to improve performance. The effectiveness of these changes vary. The first of these was a clear win, the second depends on zone usage, and the third has no effect.

#### 5.3.1 Shortened Free Page Lookup

The CMU version would walk through each entry in the `zone_page_table` looking for pages to `kmem_free`. OSF's version of the zones code uses an 8 Mbyte `zone_map`. A machine with a `page_size` of 4K would have to walk through 2K page table entries. The original code made an effort to decrease the number of examined pages by limiting the address range to that used by the continuous space allocator, `zget_space`. The problem with that method of limiting the search range was that collectable zones do not use the continuous allocator. Therefore, if all zones are collectable, the variables which defined the valid address range were never valid.

To decrease the number of iterations a pointer was added to the page table entry definition. When a page is available for collection its page table entry is linked by the `zone_add_free_page_list` routine. into a local list of collectable pages. When the search for collectable pages is over the second loop will then walk down this newly created free list, calculate the page address from the address of the page table entry, and `kmem_free` the appropriate page. Some informal measurements were made and it was discovered that the size of this free page list was typically less than ten pages.

There is a little trickiness to the first loop's free page test. Since elements may span page boundaries both endpoints of the element are tested for their collectability. If either endpoint now lies in a collectable page then the range is passed to `zone_add_free_page`. This subroutine will then add either or both of the pages to the free list. The careful reader will note that testing both endpoints is insufficient if a zone's element size is greater than twice the system page size. This is not a problem since in practice zone elements are smaller than the page size. Subsystems which desire multiple pages bypass the zones allocator in favor of direct use of `kmem_alloc`.

### 5.3.2 Lazy Evaluate Page Table Accounting

Accounting for the number of free elements per page was added to the ZALLOC and ZFREE macros. A test for the collectable flag and a call to `zone_page_in_use` became part of ZALLOC. A similar invocation of the `zone_page_free` routine was added to ZFREE. On a DECstation 3100 the original inline zone allocate and free macros were short and fast. OSF's GCC compiler generates twelve and nine MIPS instructions, respectively, for these macros. The allocate count assumes the zone has available elements and a call to the `zalloc` routine is not necessary. All counts are on a uniprocessor so no locking is involved.

The garbage collection accounting added 54 and 43 instructions to the allocate and free macros. The round trip time for an element has increased from 21 to 118 instructions; a factor of 5.6 times the original value. This is a linear increase for a RISC architecture. A CISC architecture could be further penalized by the introduction of the subroutine call.

Accounting during the `zalloc/zfree` has a detrimental effect on multiprocessor performance. There is a single lock for the zone page table. This lock is taken by all of the garbage collection routines before performing any modification of the table. This effectively single threads the zones package since this single lock must be acquired for any zone allocate or free operation. Beforehand, only the zone specific lock had to be acquired so operations on separate zones could occur in parallel.

Based on these observations, it was decided to delay the free element accounting until garbage collection was invoked. This has the beneficial effect of restoring the macros to their original sizes and removing the single lock issue. The drawback is that it now requires two passes over a zone's free list: once to calculate the `in_free_list` value for each of the elements, and once to remove the elements belonging to collectable pages.

The parallelism issue makes this change a clear win for multiprocessors but its effect is questionable for uniprocessors. This optimization effectively removes the `zone_page_in_use` subroutine call and delays the `zone_page_free` call until collection time. But it must call the latter once for every free element in the page every time `zone_gc` is invoked, whereas the original method only counted active elements. If equal cost is assigned to the two subroutine calls then for each garbage collection period the number of element round trips per page must be greater than half the number of free elements per page in order to gain any performance increase. This is possible, given that the time between garbage collections is large in comparison to element use, but there is currently no experimental evidence to bear this out.

### 5.3.3 Decrease Use of `all_zones_lock`

The `all_zones_lock` protects a zone's `next_zone` field. Rather than acquiring and holding this lock while garbage collecting all zones, this lock is only acquired when determining the next zone. (This work only applies to multiprocessors. The `all_zones_lock` is a simple lock and uniprocessors typically define out simple locks.) On the surface this is a fairly gratuitous change. The `next_zone` field is only modified when creating a new zone and zones are usually created only at system startup time. Holding an unused lock causes no harm. However, OSF/1 will be

supporting loadable device drivers and file systems. These systems will be creating their zones at load time and will therefore need to acquire the `all_zones_lock`. It is not inconceivable that a heavily loaded system would delay or block the loading of a device driver.

## 5.4 Deployment and Documentation

Garbage collection was tested extensively on two of the three OSF reference platforms, the DECstation 3100 and the Multimax, with only a small number of zones marked as collectable. When the code was proven to be reliable the default zone flags were changed to mark all zones as collectable and the `#if COLLECT_ZONE_GARBAGE` definitions that isolated the collection changes were removed. The system was tested again and the work was released to the OSF development community for general use.

Only a few problems occurred after the release. The new zones code turned up some cases of OSF subsystems attempting to use pointers that they had already freed. These problems became painfully obvious when a returned pointer belonged to a zone page that was garbage collected. The DECstation would TLB miss on any subsequent accesses of that element.

The last step was documentation. Comments were added to the zones code to explain the collector. Also, a set of manual pages were added to OSF/1 internals guide to document the zone routines and structures.

## 6. Performance

A simple performance test was done when the code merge was complete. The primary intent of this test was to quickly verify that no performance had been lost as a result of this work. The test consisted of timing an increasing number of parallel kernel compilations on two of the reference platforms: the Encore Multimax and the DECstation 3100. The tests showed that the addition of garbage collection did not significantly alter the parallel compilation times.

## 7. Conclusions

There is room for more work on the collector structure and policy. The zones collection policy is currently to collect every page possible whenever it is invoked. Alternative collection policies could be explored such as the use of high and low water marks for a zone's free element count. One could also consider restructuring the collector to use multiple threads for a multiprocessor. A more rigorous set of performance tests and benchmarks should be developed before pursuing this development.

Although more work could be done, the addition of zone garbage collection is a sound solution to the basic problem of unlimited zones growth. It was a very good addition to both the microkernel and OSF/1.



## References

1. Avadis Tevanian, Jr., "Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach", PhD thesis, CMU-CS-99-106, Dept. of Computer Science, Carnegie-Mellon University, December, 1987.
2. T.P. Lee and R.E. Barkley, "A Watermark-based Lazy Buddy System for Kernel Memory Allocation," Proc. of the 1989 Summer USENIX Conference, June 1989, pp. 1-13.
3. T.P. Lee and R.E. Barkley, "A Lazy Buddy System Bounded by Two Coalescing Delays per Class", Proc. of the Twelfth ACM Symposium on Operating Systems Principles, December 1989, pp.167-176.
4. S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman, "The Design and Implementation of the 4.3BSD Unix Operating System", Addison-Wesley, Reading, MA (1989) pp 28-29.



# Extending The Mach External Pager Interface To Accommodate User-Level Page Replacement Policies

*Dylan McNamee and Katherine Armstrong*

Department of Computer Science and Engineering  
University of Washington  
Seattle, Washington 98195

## Abstract

The Mach external pager interface allows applications to supply their own routines for moving pages to and from second-level store. Mach doesn't allow applications to choose their own page replacement policy, however. Some applications have access patterns that may make least recently used page replacement inappropriate. In this paper, we describe an extension to the external pager interface that allows the programmer to specify the page replacement policy as well as the backing storage for a region of virtual memory.

## 1 Introduction

An operating system attempts to be all things to all users. Because of this, sometimes compromises have to be made; performance may be sacrificed for generality, or modularity for performance. Virtual memory page replacement schemes are an example of such a tradeoff. While the LRU page replacement policy rarely misbehaves grossly, it rarely provides optimal performance for any application. For some groups of applications we may suspect that a different page replacement algorithm would outperform LRU. Unfortunately, existing operating systems have a single, fixed page replacement scheme. Thus these applications either had to suffer the consequences, or had to make use of some special operations, such as "wire" and "flush", that may be provided by the operating system to alter LRU's actions.

The Mach operating system provides a user with certain control over the paging of an application. There is an external pager interface that allows



applications to supply their own routines for moving pages to and from second-level store. Mach doesn't allow applications to choose their own page replacement policy, however. The Mach kernel chooses the page to be replaced, using an approximation to global LRU, and presents it for pageout to the pager associated with that page. The ability to specify backing store makes Mach's virtual memory system very flexible. Having gone as far as it has, it is a natural extension for Mach to give external pagers control over the page replacement policy for the pages under their control, since in many cases applications should know more about the patterns of access to their pages than the kernel. That is the accomplishment of the work described in this paper.

The next section motivates the extensions to Mach that will provide this facility. Section 3 summarizes Mach's virtual memory system. Section 4 discusses the various design decisions we made in the design of a user level page replacement system. Section 5 presents the structure of a system that uses our extensions. Section 6 presents a summary of the implementation status and preliminary performance results.

## 2 Motivation

Early on in the history of virtual memory research, various page replacement policies were suggested, with least recently used (LRU) being the most commonly discussed (and implemented) policy for choosing pages to remove from physical memory. Since then, most page replacement policies implemented by operating systems have been variants of LRU.

Understanding how to structure and use memory efficiently in parallel and distributed environments is still very much a topic of research. Features such as coresident task forces [Ousterhout 84], data replication, and coherence for distributed shared memory could be facilitated by extensions to the virtual memory system. Further, the growing gap between processor speeds and the speed of memory and I/O makes the choice of page replacement policy an important issue even in a uniprocessor environment. Although the trend toward larger memories will reduce page faults to some degree, those that still occur are becoming proportionately more expensive to service; therefore a paging policy that could further reduce page faults is potentially beneficial. A platform that allows different paging policies to be easily implemented and evaluated would be helpful in light of the demands

of these new developments.

Through the external memory management interface, Mach already lets the programmer choose the backing store for a memory object<sup>1</sup> by selecting an external pager. Thus external pagers can be used to implement distributed backing store and shared virtual pages. This paper presents an extension to this interface that allows the programmer to implement the page replacement policy for a memory object as well.

A garbage collector is an example of a specific application that could benefit from the ability to implement its own page replacement policy. Rather than use the default, LRU, an external pager could maintain a prioritized list of pages to page out, with high priority pages being the pages it wishes to keep cached. The pages of text and data belonging to the garbage collector itself would have high priority, and the pages of data it has finished collecting would have lower priority. When asked to remove pages from physical memory, the pager would suggest garbage collected pages before its own private data, even though some of the private pages could be older than recently collected pages. This policy is certainly preferable to LRU, which would have kept the garbage collected pages in longer, since they were more recently touched.

Another application area that could make good use of the ability to choose its own page replacement strategy is databases. Stonebraker [Stonebraker 81] has observed that database systems access pages in a manner that makes LRU replacement inappropriate. Locality, a concept upon which the LRU policy is based, tends not to be as strong in the access patterns of database applications [Kearns & DeFazio 89]. An implication of this observation is that recency of access may not be the appropriate reference property upon which to base a page replacement scheme for database applications. Using user-level page replacement, an external pager could implement a strategy more appropriate for database accesses.

As a final motivating example we discuss Camelot [Eppinger 89], a transaction facility built on top of Mach using Mach's existing external pager interface. Transaction managers need to ensure that changes to virtual memory are also recorded on permanent storage. Since *hot pages*<sup>2</sup> will never be

<sup>1</sup>A memory object is an abstract object representing a collection of data bytes on which several operations (e.g. read, write) are defined [Young et al. 87]. Memory objects are discussed more fully in the next section.

<sup>2</sup>A *hot page* is a page that is used (either written or read) frequently. A dirty hot page

least recently used, the LRU scheme will never flush them on its own. In order to prevent the number of log entries from growing without bound, Camelot must explicitly check for hot pages and periodically force Mach to flush them. With our extension to the Mach external memory management interface, Camelot could define an external pager that performed this action. Along with providing more modular system design, performance could be improved if a page replacement policy could be found that outperforms LRU in this setting. This example suggests that as experience grows, programmers could develop libraries of external pagers, each suited for certain types of applications or memory objects. As an example, an external pager for transaction systems could pageout dirty pages before clean pages, and unreferenced clean pages before referenced clean pages.

### 3 Mach Virtual Memory

We chose to use the Mach operating system for our implementation because the Mach abstraction of memory objects made the implementation easier than it would have been under most other operating systems. This section describes the memory object abstraction, and how it can be used to write external pagers<sup>3</sup>.

The channels of communication between the Mach kernel, an application and an external pager are shown in Figure 1.

A detailing of the relevant calls between kernel and user level is given in the Appendix. Each call is classified (using the letters in Figure 1) by the communication channel along which it passes. The application and its external pager may have a line of communication, if the pager's paging policy is adaptive, for example, but we have left this out of the diagram.

A memory object is an abstraction for a region of virtual memory in Mach. Physical memory is used to cache the contents of memory objects. Applications can map one or more memory objects into their address spaces, allowing them to access the data associated with those memory objects. If a task attempts to access data that is not currently cached, the kernel

---

has a log of all updates to it since its last write. Paging out a hot page to disk erases the log for that page [Eppinger 89].

<sup>3</sup>The terms external memory manager, external pager, and memory object all refer to a server implementing the external memory management interface [Young 89].

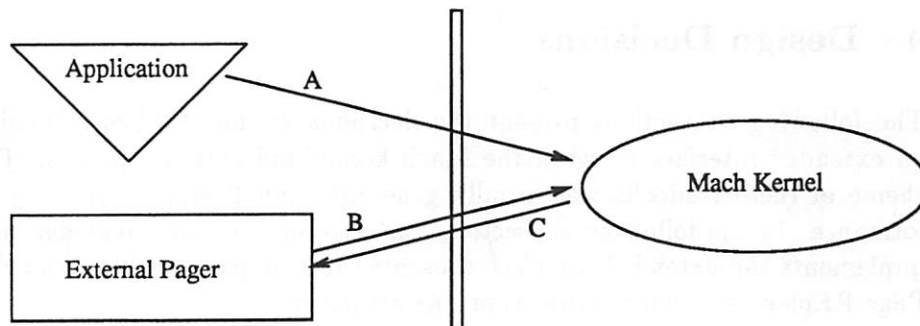


Figure 1: Communication Channels

sends a message to the external pager responsible for that region of memory, requesting that the data be retrieved from backing store. When the kernel removes dirty pages from memory, it makes requests of the external pagers responsible for those pages to write their contents to the appropriate backing store.

The `vm_allocate` call allocates a region of virtual memory backed by the default pager. `Vm_map` maps an external memory object into a task's address space. When a page fault occurs on a page backed by an external pager the kernel calls `memory_object_data_request`, which the pager is responsible for implementing. Once the pager has retrieved the data, it returns a pointer to the data to the kernel by calling `memory_object_data_provided`.

When a dirty page managed by an external memory manager needs to be removed from physical memory, the kernel calls `memory_object_data_write`. An external pager is expected to store the data it receives in a `memory_object_data_write` call and free the associated page so that it may be reused.

Communication between memory objects and the Mach kernel is through Mach RPC. RPC stubs are generated using the Mach Interface Generator (MIG) [Draves et al. 89]. This means that a memory object may be transparently connected to a remote kernel. Any number of tasks may map a particular memory object. Two tasks on the same machine may map the same portion of a memory object in order to share virtual memory. The same cached pages of physical memory may then be accessed by both tasks.

## 4 Design Decisions

The following subsections present the decisions we faced when designing an extended interface between the Mach kernel and external pagers. The theme of these tradeoffs was usually generality and flexibility versus performance. In the following subsections, an external memory manager that implements the extended interface presented in this paper will be called a Page REplacing Memory Object, or PREMO pager.

### 4.1 Direction of Information Flow

To allow an external pager to make page replacement decisions, it must have access to information about the pages it is managing. Many page replacement policies require page reference information to maintain the pages in next to replace order. Other information a pager may require are a page's dirty/clean status and whether it is active or inactive.<sup>4</sup> Since this information is currently available only to the kernel (via the *pmaps*), it was necessary to add a line of communication between the kernel and the pager. The direction of information flow could either be from the pager to the kernel or vice versa. This choice involves a tradeoff between performance and flexibility.

In the first alternative (information flows from pager to kernel), for each of its regions of virtual memory, the external pager selects from a list of kernel-implemented page replacement algorithms and communicates its choice to the kernel. After the initial communication of the policy choice, the kernel can maintain a list of the memory object's pages in the order prescribed by the page replacement policy without any further communication with the pager. Assuming a good implementation of each policy, this option has the greatest performance potential. The main problem with this approach is that it lacks flexibility. An application that requires a page replacement policy not currently supported by the kernel has to settle for a less efficient policy. The pager has no more information about its pages than it did before, which would prevent it from making adaptive paging deci-

---

<sup>4</sup>In Mach, a page is "active" when a reference to it exists in some task's *pmap*. A *pmap* is Mach's machine independent view of the information the hardware uses to perform virtual to physical address translation. Inactive pages (pages with no current references) may be reactivated if they are referenced, or are eventually sent to their pager to be written to backing store and freed.



sions. Pagers would have some control over policy, but none over mechanism. Clearly, this is not a good choice if experimentation with new replacement policies is a goal.

The alternative is to have the kernel keep the pager informed about page usage, allowing the pager to implement its own policy. This increases the amount and frequency of data flow between the kernel and the pager, which would decrease performance relative to the same (static) policy implemented within the kernel. On the other hand, a PREMO pager can implement a dynamically adaptive policy<sup>5</sup> only if the user-level external pager manages the replacement order of its pages. In addition, putting both policy and mechanism at the user level is more in keeping with the philosophy of the existing Mach external pager interface. Further, it allows applications to implement whatever page replacement policy seems natural, rather than have to settle for a “canned” policy provided by the kernel. For all of these reasons, the direction of information flow in our implementation is from the kernel to the memory object.

## 4.2 Communication Methodology

In dealing with the question of how to transfer information about pages from the kernel to the pager, simplicity of design and performance were the main criteria. We thought about allowing external pagers to have access to the page usage data kept in Mach’s kernel-level pmap data structures, but ruled that out because of the protection issues inherent in moving data structures required by the kernel into user-level memory. The second option was to have the kernel maintain a copy of the desired information in a region of memory shared between the external pager and itself. This option was investigated but discarded as being too complex for the initial implementation. The final option, and the one we chose, was to have the kernel communicate with memory objects using Mach RPC. Using MIG with Mach RPC was appealing primarily because of its simplicity and modularity.

---

<sup>5</sup>An adaptive paging policy can adjust to the memory needs of the application. An example is the working set model [Denning 70] of page replacement. Its goal is to reduce the thrashing that occurs when a task cannot acquire enough physical memory to run efficiently. Other adaptive policies could be implemented with PREMO pagers to achieve high performance in less dire circumstances.

### 4.3 Communication Content

Certain page replacement policies may require more than just page reference information. For this reason, we include the ability for a PREMO pager to select which subset of the following set of page usage data the kernel should maintain on its behalf: referenced, active/inactive, dirty/clean. This selection is made when a memory object is initialized by the kernel. Future work in this area would be to identify and incorporate other information external pagers could want.

### 4.4 Communication Frequency

Page reference information is currently provided to the Mach kernel directly by the hardware. The implementation varies somewhat with the particular architecture (e.g., VAXes don't have reference bits, so information about when a page is touched has to be simulated by the VAX-dependent parts of Mach). In turn, we could have the kernel provide page usage information to the pager as often as each page access, or as infrequently as not at all. Clearly either extreme has major drawbacks. Providing information too infrequently would leave the memory object unable to order its replacement lists with reasonable accuracy. Too frequent updates would reduce system throughput because of communication overhead.

As a compromise, our implementation provides page reference information to a PREMO pager each time a page is activated or deactivated. This information is provided via the `memory_object_page_info` call from the kernel. The PREMO pager's information is thus updated at a sufficiently fine granularity, and communication isn't so frequent that it bogs down the kernel. Other page usage information, such as whether a page is dirty or clean, is similarly provided at a fine enough granularity that the memory object can have reasonably complete information about its pages, while not significantly affecting system performance.

## 5 Structure of a System Using PREMO Pagers

In an application that uses an external pager, the existing pager can be replaced by a PREMO pager without having to make any changes. The application acquires a communication port from the memory object server



and links the memory object to a region of virtual memory using the kernel call `vm_map`. During the processing of the `vm_map` call, the kernel calls `memory_object_init` to let the memory object server know it will be responsible for this region of virtual memory. Up to this point the interaction is identical to the standard external pager initialization. As the PREMO pager initializes itself in `memory_object_init`, it lets the kernel know it will be making its own paging decisions by calling `vm_map_premo`. At this point everything is properly initialized; the region of virtual memory defined in the `vm_map` call is backed by a page replacing memory object.

When the Mach kernel needs to reclaim some physical pages, it looks at the page at the top of the kernel's replacement list.<sup>6</sup> If that page is handled by a non-PREMO pager, the kernel deactivates the page. If the page belongs to a PREMO pager, the page is instead moved to the tail of the replacement list (to keep it from being repeatedly selected), and the PREMO pager is asked to select one or more pages to relinquish. This information is passed to the kernel by calling `memory_object_hints_provided`. This implements a two-level page replacement scheme. The choice of which memory object must give up a physical page is determined by the kernel-maintained LRU ordering of pages caching the contents of memory objects. When the selected memory object is managed by a PREMO pager, the choice of which of the pages will be relinquished is made by the PREMO pager. One can envision a scheme whereby both levels of this two-level approach are variable. How one would choose a suitable policy and mechanism for the first level is an interesting problem, however it is beyond the scope of this paper.

## 6 Implementation and Performance

The PREMO pager interface has been incorporated into Mach 2.5 running on a MicroVAX II. Synthetic applications with known paging behavior were coupled with PREMO pagers to verify the correctness of the implementation. We are currently investigating incorporating PREMO pagers into real applications.

There are two points to consider when deciding whether to incorporate PREMO pagers into an application. The most important consideration is whether an application's memory access patterns make LRU or its variants

---

<sup>6</sup>Mach presently uses second chance FIFO as an approximation to LRU.

inappropriate. PREMO pagers could potentially improve the performance of such applications by providing an alternate page replacement policy that significantly reduces paging.

The second consideration is the communication overhead incurred by using PREMO pagers, and whether the gains due to reduced paging outweigh the added overhead. To measure the overhead of using PREMO pagers, we compared the elapsed runtime of our synthetic benchmark program using a PREMO pager that implements the same replacement policy as the kernel, to the same program using the kernel's default pager. Preliminary measurements indicate that the overhead associated with using a PREMO pager adds approximately 10% to this application's runtime.

To get an idea of the increase in performance made possible by PREMO pagers, we wrote a PREMO pager that took advantage of the benchmark program's memory access patterns to reduce page faults. This pager reduced the number of page faults by 15%. While these results are encouraging, it is important to investigate the performance of real applications running with PREMO pagers. We expect to present more complete performance figures at the workshop.

## 7 Conclusions and Future Work

The Mach external pager interface allows pagers to specify backing storage for memory objects. Our extension to this interface gives external pagers the capability to manage the paging behavior of their resident pages. Using this extension, paging policies that are more appropriate for special classes of applications, such as database and transaction systems, mapped files, and garbage collectors, may be implemented. Kathy Armstrong is presently investigating the use of alternate paging policies for mapped files. Other work includes investigating the integration of page replacing memory objects into other classes of applications in a distributed environment.

The present implementation of the interface allows potentially misbehaving pagers to monopolize physical memory. As the implementation matures we will add security checks to ensure that each pager minimally fulfills its paging responsibility. If, as our experience grows, communication costs turn out to be a limiting factor to performance, we may consider implementing the shared memory communication method.

## 8 Acknowledgments

We wish to thank Tom Anderson, Brian Bershad, Eric Koldinger, Ed Lazowska, Brian Pinkerton, and John Zahorjan for their helpful comments. We also appreciate the efforts of Jan Sanislo and Eric Lundberg in obtaining the source code and hardware necessary for our implementation.

## 9 Appendix – Partial interface between an external pager and the kernel

### From kernel to memory object

`memory_object_hint_request` requests hints from a page replacing memory object. Type C (see Figure 1.)

`memory_object_page_info` provides requested information about a page to a page replacing memory object. Type C.

`memory_object_init` serves notice to a memory object that Mach has been asked to map a given memory object into a task's virtual address space. Type C.

`memory_object_data_request` is a request for data from the specified memory object. This data is returned to the kernel with the `memory_object_data_provided` call. Type C.

`memory_object_data_write` provides the memory object with data that has been modified while cached in physical memory. Once the memory object has written this data to second level store, it is expected to deallocate the physical memory, using `vm_deallocate`. Type C.

### From user level to kernel

`vm_map` maps a region of a memory object into a task's address space. Type A.

`vm_map_premo` informs kernel this is a page replacing memory object. Type B.

`memory_object_hints_provided` tells kernel which cached pages may be discarded. Type B.

`memory_object_data_provided` provides data for a region of virtual memory. Type B.

`memory_object_data_error` indicates that the memory object cannot return the data requested for the given region. Type B.

`memory_object_data_unavailable` indicates that this memory object does not have data for the given region. Type B.

`memory_object_lock_request` allows a memory object to make special requests of the kernel. These include write back, flush, lock, and unlock. Type B.

`memory_object_set_attributes` controls how Mach uses the memory object. The memory object can request that Mach keep its pages in physical memory, even when the pages are no longer mapped into any virtual address space. Type B.

## References

- [Baron et al. 89] Robert V. Baron, David Black, William Bolosky, Jonathan Chew, Richard P. Draves, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael Wayne Young. "Mach Kernel Interface Manual" Technical Report, Department of Computer Science, Carnegie-Mellon University, June 1989.
- [Denning 70] P. J. Denning, "Virtual Memory," *Computing Surveys*, Volume 2, Number 3, 1970.
- [Draves et al. 89] Richard P. Draves, Michael B. Jones, Mary R. Thompson. "MIG - The MACH Interface Generator" Technical Report, Department of Computer Science, Carnegie-Mellon University, November 1989.
- [Eppinger 89] Jeffrey L. Eppinger. "Virtual Memory Management for Transaction Processing Systems." PhD thesis, Carnegie-Mellon University, 1989.
- [Forin et al. 89] Alessandro Forin, Joseph Barrera, and Richard Sanzi. "The Shared Memory Server" *1989 Winter USENIX Conference*, 1989.

- [Kearns & DeFazio 89] John P. Kearns and Samuel DeFazio. "Diversity in Database Reference Behavior" *Performance Evaluation Review*. Volume 17, Number 1, 1989.
- [Levy & Eckhouse 88] Henry M. Levy and Richard H. Eckhouse, Jr. *Computer Programming and Architecture: The VAX*. 2nd Ed. Digital Press, 1988.
- [Ousterhout 84] John K. Ousterhout. "Scheduling Techniques for Concurrent Systems" *Proc. 3rd IEEE International Conference on Distributed Computing Systems*, 1984.
- [Peterson & Silberschatz 85] James L. Peterson and Abraham Silberschatz. *Operating System Concepts*. Addison-Wesley Publishing Company, Reading, MA, 1985.
- [Stonebraker 81] Michael Stonebraker. "Operating System Support for Database Management" *Communications of the ACM*. Volume 24, Number 7, 1981.
- [Tevanian et al. 87] Avadis Tevanian, Richard Rashid, Michael W. Young, David B. Golub, Mary R. Thompson, William Bolosky, Richard Sanzi. "A Unix Interface for Shared Memory and Memory Mapped Files Under Mach" Technical Report, Department of Computer Science, Carnegie-Mellon University, July 1987.
- [Young et al. 87] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System" *Proc. 11th Symposium on Operating Systems Principles*, 1987.
- [Young 89] Michael Young "Exporting a User Interface to Memory Management from a Communication-Oriented Operating System." PhD thesis, Carnegie-Mellon University, 1989.





# Mach on a virtually addressed cache architecture

*Chia Chao, Milon Mackey, Bart Sears*

*cchao@hplabs.hpl.hp.com, mackey@hplabs.hpl.hp.com, sears@hplabs.hpl.hp.com*

Hewlett-Packard Laboratories  
1501 Page Mill Road  
Palo Alto, CA 94304

## Abstract

Mach shares memory between tasks by virtual address aliasing. This means a physical page can be mapped to more than one virtual page at a time. This works well on hardware architectures that use a physically addressed cache, but can cause some problems for architectures that use a virtually addressed cache resulting in less than best possible performance. This paper describes four experiments that were tried to improve the performance of Mach running on a machine with a virtually addressed cache. The experiments were implemented and measured against a collection of benchmarks. The measurements were then compared to those taken on a "base" implementation of Mach on the same hardware to get a clear idea of the performance benefit of each experiment.

## 1 Introduction

We ported Mach [Tevanian 87a] [Tevanian 87b] to a machine which uses a virtually addressed cache. While doing this, we noticed that the Mach machine independent code made some assumptions based on a physically addressed cache. These assumptions caused some performance penalties on machines with a virtually addressed cache. After finishing our port, we tried four experiments to improve the performance of Mach running on a machine with a virtually addressed cache. The experiments were implemented and measured against a collection of benchmarks. The measurements were then compared to those taken on a "base" implementation of Mach on the same hardware to get a clear idea of the performance benefit of each experiment.

This paper starts by giving an introduction to the machine architecture on which the experiments were performed, a history of our effort to port Mach, and an overview of the performance problems caused by virtual address aliasing. It then describes the experiments, the measurements, and our conclusions.

## 2 PA-RISC

Precision Architecture (PA-RISC<sup>1</sup>) is Hewlett-Packard's reduced instruction set (RISC) architecture. It is described in [Mahon 86], [Lee 89], and [HP 89]. We give a summary here.

PA-RISC provides one *global* 64 or 48 bit virtual address space<sup>2</sup> in which the kernel and all tasks reside. Except for protection restrictions, any task can read or write any virtual address in the global address space. Protection is on a virtual page basis and is done through the conventional mechanisms of access rights, privilege levels (PA-RISC has four), and access ids. Access rights are used to specify the types of access (*r*, *w*, *x*) that are permitted on a page. Privilege levels and access ids limit access to a page to only those tasks that are running at a privilege level in the range specified for the page and who are holding a protection id that matches the access id on the page. Zero is a special access id. If a page has its access id set to zero, only the access rights and privilege level checks are done before granting access to the page.

<sup>1</sup>PA-RISC used to be called HP-PA.

<sup>2</sup>The PA-RISC architecture specifies a 64-bit limit to the size of a virtual address. All current implementations of the architecture use 48-bit virtual addresses.

The PA-RISC Architecture recommends using an *inverted* page table to record the virtual to physical page mappings. The PA-RISC designers wanted to support large sparsely populated address spaces. In this situation it is not practical to use a “standard” page table containing an entry for each virtual page, since most virtual pages may never be used. In contrast, an inverted page table contains an entry for each physical page. The entry corresponding to a physical page indicates whether the page is mapped and, if so, it specifies the virtual page to which it is mapped. To translate from a virtual to a physical address a hash table is used.

PA-RISC uses a Translation Lookaside Buffer (TLB) and a virtually addressed write-back cache. A TLB is a hardware mechanism used to translate a virtual memory address into a physical memory address and to perform the page protection checks. The PA-RISC designers chose to use a virtually addressed cache so that the TLB access and cache lookup could proceed in parallel, allowing the CPU cycle time to be shortened. As CPU cycle times become shorter and shorter, optimizations such as this increase in importance and make it easier to build very fast processors. On a physically addressed cache architecture, the cache lookup can only begin after the TLB has translated the virtual address into a physical address which can then be used to do the cache lookup.

In PA-RISC the cache is normally given a 48 or 64 bit virtual address. When physical addressing mode is enabled accesses to memory still go through the cache. In this case, the the upper 16 or 32 bits of the address are set to zero. The physical address is used for the lower 32 bits of the address.

Since PA-RISC provides one global virtual address space, the “right” way to share memory between tasks on PA-RISC is to have all the tasks sharing the memory to access the memory using the same common range of virtual addresses. Also notice that since all tasks run in one global virtual address space on PA-RISC there is no need to flush the cache on a context switch from one task to another.

### 3 History of Tut

Tut is the name of our project to port Mach to PA-RISC. We give a short summary of our approach and results here.

Rather than porting Mach to PA-RISC by first porting Berkeley 4.3 and then turning on the Carnegie Mellon (CMU) Mach `ifdefs`, we decided to “put” Mach into HP-UX [Clegg 86], Hewlett-Packard’s version of UNIX. We did this in two steps. First we replaced HP-UX’s virtual memory subsystem with Mach’s virtual memory subsystem and then we replaced HP-UX’s process management with Mach’s tasks and threads. This process left the rest of HP-UX for the most part unchanged. We took this approach because we wanted it to be easy to move programs from HP-UX to Tut and we thought that this approach had a good chance of preserving most of HP-UX’s functionality intact. We also liked the HP-UX kernel debugger [HP 87] and didn’t want to port it to a Berkeley 4.3 base. In addition, we wanted to use the HP-UX test suites to test our implementation.

The version of HP-UX that we started with, HP-UX 2.0, was based on Berkeley 4.2 with some Hewlett-Packard extensions to add real time capability and to make it SVID<sup>3</sup> compliant. The version of Mach that we started with was Mach 2.0. HP-UX 2.0 had a vnode file system, but Mach 2.0 did not. Consequently Mach 2.0’s inode pager would not work with the HP-UX 2.0 file system. This problem was solved by obtaining a working vnode pager from an unreleased snapshot of Mach taken sometime between Mach 2.0 and 2.5. Once Mach 2.5 became available, we also replaced the 2.0 version of `vm_fault()`, the routine used in the Mach virtual memory subsystem to handle page faults, with the 2.5 version of `vm_fault()`. This was done to fix some defects that existed in the 2.0 `vm_fault()` algorithm.

When we were done, we had a Tut kernel that behaved for the most part like HP-UX 2.0 with new Mach functionality consisting of Mach virtual memory management, a vnode pager, tasks, threads, and Mach IPC. The Tut kernel does not have working external pagers and still uses the standard UNIX file system buffer cache, since external pagers and Mach’s no buffer cache option were not yet working in Mach 2.0. The Tut kernel also does not support all of the HP-UX real time extensions (`rtprio` is not supported) nor System V shared memory, which are standard features of HP-UX 2.0.

<sup>3</sup>HP-UX passes the System V Interface Definition tests.

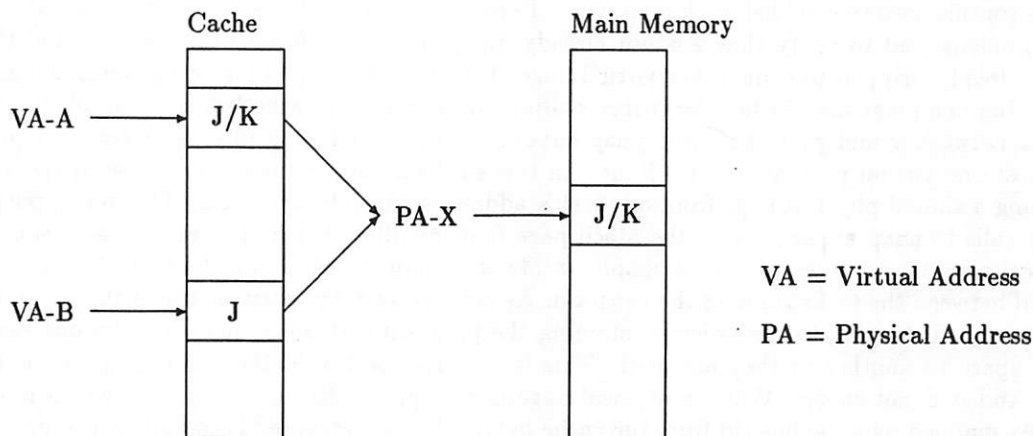


Figure 1: Alias problems with virtual caches

The Tut pmap module uses an inverted page table to record the virtual to physical page mappings like HP-UX 2.0. We succeeded in our original goal of program portability. As long as an HP-UX executable doesn't use any HP-UX features that we broke and doesn't read any data structures from `/dev/kmem` that were changed during the port, it will run on a Tut kernel with no changes.

Tut gave us an environment in which to experiment with performance enhancement techniques for Mach running on hardware using a virtually addressed cache. Based on suggestions by Jim Hays of the UNIX Kernel Lab (UKL) and David Black of CMU, we set up four different experiments to improve the performance of Mach on such architectures. Before describing the experiments, we give some background as to why Mach can have performance problems when running on a machine that uses a virtually addressed cache.

## 4 How Mach does sharing

Mach shares memory between tasks by virtual address aliasing. This means a physical page can be mapped to more than one virtual page at a time. This works well on hardware architectures that use a physically addressed cache, but can cause some problems for architectures that use a virtually addressed cache resulting in less than best possible performance.

To see why virtual address aliasing causes problems for a virtually addressed cache, consider the case in Figure 1 where the virtual addresses A and B are both mapped to physical address X. On the first read of address A the value J is copied to the cache location specified by A. If the memory location A is now changed to K, the cache is changed, but the actual value in memory is not changed until the dirty cache line is written back to memory. Any reads of location B will now get the J instead of the new value K. Note that even if a write-through cache is used (where all writes go immediately to memory) there are still problems. If both locations A and B are read and then location A is changed to K, even if this is written back to physical memory, any read of B will get an incorrect value since the cache entry for location B contains J. A physically addressed cache does not suffer from these problems because virtual addresses A and B are translated into the same physical address that is used to access the cache, and so access the same cache line.

The problems caused by virtual address aliasing do not mean that Mach cannot run on a machine using a virtually addressed cache. But, they do mean that Mach running on such a machine can suffer a performance penalty unless care is taken. An obvious solution to the virtual address aliasing problem is to prevent all virtual address aliasing at the hardware level. This solution can be implemented in the pmap module, the machine dependent portion of the Mach virtual memory subsystem. The pmap module exports a routine called `pmap_enter()` that is used by the Mach machine



independent code to establish a mapping between a virtual page *v* and a physical page *p*. This is the only routine used to establish such mappings. To prevent virtual address aliasing, `pmap_enter()` can be implemented to verify that *v* is not already mapped to some other physical page and that *p* is not already mapped to some other virtual page. If any of these mappings exist, `pmap_enter()` removes the mappings and flushes the corresponding pages from the cache before it establishes the mapping between *v* and *p*. In this way `pmap_enter()` ensures that each physical page is mapped to at most one virtual page at a time. Hence, in this solution memory is shared between tasks by remapping a shared physical page from one task's address space into another's. This remapping is done by calls to `pmap_enter()` from the Mach page fault handler `vm_fault()` as the tasks sharing a physical page try to access it. For example, in Mach an object containing the text of a program is shared between the tasks running the program. As context switches occur between the tasks, the physical pages belonging to the object containing the program text are remapped from one task's address space to another as they are used. This is a costly solution if the remapping cost is not cheap. And it is not cheap. When a physical page is remapped, the virtual page to which it was originally mapped must be flushed from the cache before the new mapping is established to prevent virtual address aliasing. In addition, there is the extra overhead of processing the traps that result as the shared pages are faulted back and forth between the tasks. None of this overhead exists on a physically addressed cache architecture.

It is worthwhile trying to improve upon the "avoid all virtual address aliasing at the hardware level and do all sharing by remapping" solution just described. We now begin our discussion of the experiments we tried to improve upon this "base" solution.

## 5 The experiments

Initially the Tut `pmap` module was implemented using the simple minded solution described in the previous section: all virtual address aliasing at the hardware level was avoided and all sharing of memory between tasks was done by remapping. Four experiments were tried to improve the performance of this initial "base" implementation. Two of the experiments, shared text and read-only aliasing, address the main issue of efficiently sharing memory between tasks. Both of these experiments optimize the important special case of sharing read-only memory between tasks by avoiding remapping. Sharing writable memory between tasks is harder to optimize and was not addressed by our experiments (see Section 9). The two other experiments, `pmap_pre_enter()` and lazy mapout, address a couple of other minor performance issues that result when running Mach on a virtually addressed cache architecture.

### 5.1 Shared text

The shared text experiment addresses the issue of sharing program text efficiently between tasks. Many tasks may execute the same program at the same time. It is foolish to have one copy of the program text in physical memory for each task. In some manner it should be possible to share one copy of the program text in physical memory between those tasks that are using it. In Mach this is done by having the tasks share one copy of the program text in physical memory through virtual address aliasing.

In the base implementation of Tut, text was shared by remapping the physical pages containing the text between the tasks using them. The shared text experiment takes advantage of the one global virtual address space provided by PA-RISC to eliminate all remapping of text pages shared between tasks. The idea is to map the text object to a single range of virtual addresses in the global virtual address space, and to have all tasks sharing the text object to access it using that same range of virtual addresses. This not only eliminates virtual address aliasing at the hardware level, it eliminates it at all levels. This is the "right" way to share memory on PA-RISC and was the method used by HP-UX 2.0 to implement shared text. The original HP-UX 2.0 implementation of shared text was destroyed when we replaced HP-UX's virtual memory management subsystem with Mach's. This method totally eliminates the virtual address aliasing performance problem for text



objects. However, it does not improve the performance of any other type of shared memory between tasks, for which the simple solution of remapping is still used.

To implement shared text in this manner required changes to the Mach machine independent code. A `pobject` pointer was added to the `vm_object` structure. The `pobject` pointer points to a `pobject` structure that is associated with the object. The `pobject` structure provides a handle for machine dependent information to be associated with an object. If there is no machine dependent information, the `pobject` pointer can simply be a null pointer.

In our implementation of shared text, an object has an associated `pobject` structure only if it is being used as text. If an object is not being used as text, its `pobject` pointer is null. The `pobject` structure is used to remember the range of virtual addresses to which the text object was mapped. When an executable file is loaded via `exec()` for the first time, a text object is created and the range of virtual addresses to which it is going to be mapped is selected. Information describing that range is stored in the `pobject` structure associated with the object. This information in the `pobject` structure is used to initialize the `pmap` structure<sup>4</sup> of a task executing the text so that it can access the range of virtual addresses to which the text object was mapped. Since this information is kept with the object, it is available as long as the object exists even if no tasks are currently using the text object and it has migrated to Mach's object cache (see Section 5.4). If it is retrieved from the object cache, the information in the `pobject` structure can be used once again to initialize the `pmap` structure of the task using the text.

Besides the addition of `pobject`, the changes to the Mach machine independent code were minor modifications to some of the routines in `vm/vm_object.c` and the addition to that file of some new routines to manipulate `pobject` structures.

This method of sharing text objects will only be of benefit to other virtually addressed cache architectures that provide one global virtual address space like PA-RISC. However, the idea of being allowed to associate machine dependent information with an object may be more generally applicable.

Besides the global virtual address space provided by PA-RISC, another feature of the PA-RISC architecture made shared text easy to implement. PA-RISC provides a *short pointer addressing mode* which is an efficient way to generate the 48 or 64 bit virtual address required by the architecture. When using short pointer addressing mode the program generates a 32-bit virtual address. The two upper bits of this address select a *space register* that provides the upper 16 or 32 bits of the virtual address. These 16 or 32 bits provided by the space register are concatenated with the original 32-bit short pointer address to produce the required 48 or 64-bit virtual address. This scheme effectively divides the  $2^{32}$  byte short pointer address space up into four quadrants of  $2^{30}$  bytes each with a corresponding space register for each quadrant. In Tut, the text of a program is always mapped into the first quadrant of the short pointer address space belonging to a task. To share text between tasks, each task needs only to have the same value inserted into the space register corresponding to the first quadrant of the short pointer address space.

The measurements presented in Section 7 show that shared text provides a good performance improvement.

## 5.2 Read-only aliasing

Read-only aliasing makes the sharing of read-only memory between tasks more efficient. It does this by allowing virtual address aliasing of read-only memory at the hardware level. Since the shared memory is read-only, virtual address aliasing at the hardware level causes no problems as long as all aliases are purged from the cache if the memory happens to become writable, in which case the standard remapping solution is used.

Read-only aliasing is more general than shared text. Shared text allows program text to be shared efficiently between tasks. Read-only aliasing not only improves the efficiency of shared text, it allows any read-only memory to be shared efficiently between tasks. An example of shared read-only memory, that is not shared text, is copy-on-write shared memory. Copy-on-write memory

<sup>4</sup>The `pmap` structure contains the machine dependent information that is used to define the address space of a task at the hardware level. Each task has its own `pmap` structure.

results from the `fork()` operation because the child process inherits the parent's address space copy-on-write. The parent and child task share the physical pages containing the program data and stack read-only. When one of them tries to write to one of these shared pages, both parent and child are given their own private writable copy of the page.

A possible disadvantage of read-only aliasing is that it uses the cache and TLB less efficiently than shared text. For example, shared text will cache at most one copy of a text object in the cache, but read-only aliasing may cache as many copies as there are tasks executing the text, since each task maps the text to a different virtual address range. Similarly, shared text needs only one TLB entry to map a page of shared text. But read-only aliasing will use as many TLB entries as there are virtual aliases for the page.

The implementation of read-only aliasing only required changes to the `pmap` module. No changes to the Mach machine independent code were needed or made. The changes to the machine dependent data structures were quite extensive. In the brief overview of PA-RISC, we mentioned that PA-RISC uses an inverted page table. This data structure can only remember one virtual to physical translation per physical page. To implement read-only aliasing, this data structure must be able to remember multiple virtual to a physical page translations and it was changed to do just that.

The measurements presented in Section 7 show that read-only aliasing provides a good performance improvement.

### 5.3 `pmap_pre_enter()`

The `pmap_pre_enter()` experiment addresses a performance issue relating to page initialization. Mach is designed to work on multiprocessors and is multithreaded. When a page fault occurs, threads other than the one causing the fault may try to access the missing virtual page. None of these threads can be allowed to access the page until it has been properly initialized by the Mach virtual memory subsystem. Mach prevents such access by selecting a physical page to satisfy the fault and initializing it before it is mapped to the virtual page causing the fault.

This solution makes few assumptions about the capabilities of the Memory Management Unit (MMU) and works well with a physically addressed cache. The `pmap` routines, `pmap_zero_page()` and `pmap_copy_page()`, used to initialize physical pages, can do their work using physical addresses. Since the cache is physically addressed, the act of initializing the page will prestage the page into the cache. Later, after the physical page is mapped, access to the page will be fast since there will be few cache misses.

On a virtually addressed cache architecture the situation is different. When `pmap_zero_page()` or `pmap_copy_page()` is called to initialize a physical page `p`, the `pmap` module does not know the virtual page `v` to which `p` is going to be mapped. This information is not available to the `pmap` module until `pmap_enter()` is called to actually map `v` to `p` which is *after* `pmap_zero_page()` or `pmap_copy_page()` has been called to initialize `p`. If no changes are made to the `pmap` module interface, there are two methods that can be used by `pmap_zero_page()` and `pmap_copy_page()` to initialize `p`. One is to initialize `p` using physical addressing mode. This is the method that works well on a physically addressed cache architecture. The other method is to map `p` to a reserved virtual page that is inaccessible to kernel and user tasks and then initialize the page using virtual addressing mode. Later when `pmap_enter()` is called `p` can be remapped from the special virtual page to `v`. Both methods have drawbacks. The main one being that neither method prestages the page in the cache at the correct virtual address. That can only be done by knowing `v`, mapping `v` to `p`, and using virtual addresses in `v` to initialize `p`. Other drawbacks are that the second method requires that the range of virtual addresses used to initialize the page be flushed from the cache to prevent virtual address aliasing. The first method may also require a cache flush if memory accesses in physical addressing mode go through the cache and the cache is write-back, which is true on PA-RISC. So not only do virtually addressed cache architectures lose the prestaging effect they may also be burdened by extra cache flushes during page initialization.

To help in this situation we experimented with a new `pmap` routine `pmap_pre_enter()`. It is optional and need not be implemented for architectures for which there would be no performance benefit. It gives a hint to the `pmap` module that there is a good chance that a physical page `p` is

going to be mapped to a virtual page *v* in the near future. This information is given to the pmap module before the page *p* is initialized. This way the pmap module has the option of mapping *p* to *v* early and initializing the page using the correct virtual addresses, prestaging it in the cache. This solution can only be used if the MMU has enough functionality so that the page can be mapped by a call to `pmap_pre_enter()` but protected so that it cannot be accessed by *any* user or kernel thread before the page has been initialized and made accessible by a call to `pmap_enter()`. In our PA-RISC implementation, this is accomplished by protecting a “pre-entered” page with a access id that can never be held by a user or kernel task. The routines `pmap_zero_page()` and `pmap_copy_page()` are the only routines that can be used to access a “pre-entered” page without incurring a protection trap. They contain code to disable the access id check before they initialize the page. After the page has been initialized, they reenable the access id check before returning to their caller.

It is possible to weaken this requirement for architectures whose MMUs have less functionality as long as only *user* pages are pre-entered. In this case the MMU only needs to satisfy the weaker requirement that a pre-entered page be inaccessible to all user threads.

This experiment required some minor changes to `vm_fault()` to make it call `pmap_pre_enter()` before calling `pmap_zero_page()` or `pmap_copy_page()`. Since we do not have working external pagers in Tut we do not know how `pmap_pre_enter()` interacts with them.

`pmap_pre_enter()` may also be of benefit on a physically addressed cache machine where using physical addressing mode is expensive to initialize pages.

The measurements presented in Section 7 show that `pmap_pre_enter()` provides a moderate performance improvement.

## 5.4 Lazy mapout

Mach caches pages of text belonging to a recently used program in physical memory, anticipating that the program will be used again in a short time. If the program is run again, work is saved because much of the program’s text is already in memory and need not be reread from disk. If the program is not run again, the pages are gradually reclaimed by the Mach pageout daemon and put to other uses just like any other pages that have not been used recently.

This feature is implemented by the Mach object cache. In Mach, each object maintains a reference count of the number of tasks using it. When the reference count becomes zero, the object is deallocated and its memory resident physical pages unmapped and put on the free physical page queue so that they can be reused, *unless* it is a text object. When a text object is no longer being used by any task, it is put into the Mach object cache. At that time its memory resident physical pages are unmapped so that they can no longer be accessed and added to the inactive page queue, from which they may be reclaimed by the Mach pageout daemon if physical memory becomes a scarce resource. A text object stays in the object cache until it is reclaimed by a task wanting to use it or it is deallocated because of old age. At that time any physical pages still belonging to the text object are added to the free physical page queue.

This works well on a physically addressed cache architecture because a page belonging to a cached object is unmapped but not flushed from the cache. If the page is reclaimed, it can be remapped and much of it may still be in cache. The situation is different on a virtually addressed cache architecture. With a straight forward approach, when a page is unmapped it must be flushed from the cache to prevent virtual address aliasing. This means a virtually addressed cache architecture has to pay a performance penalty because recently used text pages are never in the cache when a program is run again.

Lazy mapout is a technique to address this problem: the pmap module is modified to delay the unmap and flush cache operation on a virtual page as long as possible. Routines like `pmap_remove()` and `pmap_remove_all()` no longer break virtual to physical translations as they do in the “base” implementation, they simply change the protection on a page so that it can no longer be accessed by any kernel or user threads. `pmap_enter()` is the only routine that actually breaks a virtual to physical translation, flushing the corresponding virtual page from the cache. This must be done by `pmap_enter()` since an old translation must be broken before a new translation is put in its place. If the new translation is the same as the old one, `pmap_enter()` need only change the protections



on the page to allow the specified access. The modifications used to implement lazy mapout were limited to the pmap module. No changes were needed in the Mach machine independent code.

Lazy mapout by itself cannot solve the text flushing problem described above. Lazy mapout can only be of benefit in this problem if a reclaimed text page is going to be mapped to the same virtual address that it had when it entered the object cache. This is unlikely in a "base" Tut kernel, but is exactly what happens in the shared text experiment, and so lazy mapout may be thought of as an enhancement to shared text.

Lazy mapout can only be used on a machine architecture for which it is possible to protect a "lazily mapped out" page so that it cannot be accessed by any kernel or user thread. In our implementation on PA-RISC this is done by protecting a lazily mapped out page with a special access id that can never be held by a user or kernel task. We used the same technique in the `pmap_pre_enter()` experiment (see Section 5.3).

We also thought lazy mapout might offer a small performance gain on PA-RISC even without shared text, since it may allow some page initializations and page copies to be done in virtual addressing mode that would otherwise have been done using physical addressing mode. The pmap routines `pmap_zero_page()`, `pmap_copy_page()`, `copy_to_phys()` and `copy_from_phys()` were modified to be able to access lazily mapped out pages. These modifications were the same as those used in `pmap_zero_page()` and `pmap_copy_page()` in the `pmap_pre_enter()` experiment. When using lazy mapout, a physical page just removed from the free physical page queue may in fact be lazily mapped out; this means that `pmap_zero_page()` and `pmap_copy_page()` can use virtual address mode to initialize the page. The routines `copy_to_phys()` and `copy_from_phys()` are used by the vnode pager to copy between a physical memory page and buffers in the file system buffer cache during pagein and pageout operations. The Mach pageout daemon uses the pmap routine `pmap_remove_all()` to make a physical page inaccessible before the page is paged out. When lazy mapout is used, `copy_from_phys()` can use virtual addressing mode to copy the page into the file system buffer cache since the page is still really mapped. `copy_to_phys()` can also use virtual addressing mode during pagein if the target page happens to be lazily mapped out.

The measurements presented in Section 7 show that whether lazy mapout provides a performance improvement is debatable. To get a better idea we developed two special "lazy mapout" benchmarks. For those benchmarks lazy mapout does indeed provide a performance improvement.

## 6 Measurements

The modifications for each experiment were implemented so that they could be run in any combination with each other. For example, Tut kernels with shared text and lazy mapout or with read-only aliasing, lazy mapout, and `pmap_pre_enter()` can be built and work. This gave us 16 different Tut kernels to measure and the opportunity to see how the different experiments interacted with each other, enabling us to determine the most effective combinations for improving performance. In addition, the virtual page size used by the kernel can be 2K, 4K, or 8K. We were also interested in determining if there were any interesting interactions between the virtual page size used by the Mach virtual memory subsystem and the experiments. This gave us 48 different kernels to measure.

Where possible we ran the same benchmarks on the HP-UX 2.0 kernel as the Tut kernels. This allowed us to measure two things: (1) the performance effect of replacing HP-UX's 2.0 virtual memory and process management with Mach's virtual memory management system and Mach tasks and threads, and (2) the performance effects of our experiments. Our results should not be used to draw conclusions about the absolute performance of Mach or HP-UX. Our Tut kernels are based on kernel sources that are by now quite ancient, in addition, our Tut kernels were not product tested or tuned. However, our measurements do give an accurate indication of the relative change in system performance.

We continue now with a discussion of the benchmarks we used to measure the performance changes and a description of the environment in which the benchmarks were run.

## 6.1 Benchmarks

We ran four benchmarks to measure the performance changes due to the experiments. Two of the benchmarks were general in nature and two of them were developed to get more information on the performance of lazy mapout. As mentioned at the end of Section 5.4, lazy mapout did not show itself to be a clear win on the two general purpose benchmarks. The two lazy mapout benchmarks were an attempt to let lazy mapout show itself in the best possible light by benchmarking two areas in which it should be a performance win. We now give short descriptions of the benchmarks.

### 6.1.1 Andrew

The Andrew benchmark is from CMU. It was originally developed to measure and compare the performance of a standard Unix file system to that provided by Andrew, a file system developed at CMU. This benchmark runs `cp`, `grep`, and `cc`. We ran two copies of this benchmark at the same time and referred to this as "2 user" Andrew. Two user Andrew had an approximate running time on a base Tut kernel of 5 minutes. See Section 6.2 for a description of the environment in which the benchmarks were run.

The results of this benchmark cannot really be used to cross-compare machine architectures, because part of the Andrew benchmark is to compile a set of C programs, and so the elapsed time of the Andrew benchmark is highly dependent on the performance of the compiler.

When this benchmark was run on Tut kernels no paging to backing store occurred. We don't know if paging to backing store occurred when this benchmark was run on HP-UX 2.0.

### 6.1.2 MUB

The MUB benchmark attempts to simulate a workload generated in a Multi-User environment. We ran the benchmark to simulate 3 users. This benchmark had an approximate running time on a base Tut kernel of 1.5 minutes when simulating 3 users.

When this benchmark was run on Tut kernels no paging to backing store occurred. We don't know if paging to backing store occurred when this benchmark was run on HP-UX 2.0.

### 6.1.3 Lazy Mapout benchmark

This benchmark was designed to test the assertion that shared text and lazy mapout should work well together. This benchmark repeatedly ran `awk`, `nawk`, and `statistic` in a loop to measure how well text pages were being retrieved from the Mach object cache. This benchmark had an approximate elapsed running time on a base Tut kernel of 8.2 minutes and an approximate system time of 1.1 minutes.

When this benchmark was run on Tut kernels no paging to backing store occurred. We don't know if paging to backing store occurred when this benchmark was run on HP-UX 2.0.

### 6.1.4 Paging benchmark

This benchmark was designed to test the assertion that lazy mapout would help paging run faster since during a pageout operation a page could be copied to the file system buffer cache by using virtual addressing mode. We also wanted a benchmark that would force some paging to occur, since none of the other benchmarks generated any paging.

This benchmark maps a 32-Megabyte file into a section of its address space using Mach's `map_fd()` system call. It makes a modification in the file every 2K bytes. This forces a copy-on-write operation by the kernel because the file is mapped copy-on-write by `map_fd()`. The file is large enough so that it cannot all fit into main memory at once, forcing pages to be written to backing store. Once the modifications are completed, the benchmark reads a word of the file every 2K bytes to bring the pages back into memory.

This benchmark had an approximate elapsed running time on a base Tut kernel using a 4K page size of 6.2 minutes and an approximate system time of 50 seconds. On a base Tut kernel using an



8K page size it had an approximate elapsed time of 4.5 minutes and an approximate system time of 43 seconds.

## 6.2 Benchmark environment

The items listed below describe the environment in which the benchmarks were run.

- The benchmarks were run on an HP 9000 series 835, a uniprocessor implementation of the PA-RISC architecture, with 32 megabytes of main memory. This is a lot of memory. The Andrew, MUB, and Lazy Mapout benchmarks never paged to backing store when they were run on Tut kernels.
- Each benchmark was run in *single user mode*, i.e. beside the tasks executing the benchmark the only other tasks running on the test system were the `init` task and the kernel task. The network was not enabled either. This was done to minimize the effect of other users on the measurements.
- Mach (and Tut) do not have special swap areas on disc to provide backing store for virtual memory objects. Mach uses standard Unix file systems to provide backing stores, and uses the standard file system routines to page to and from backing stores.

To minimize the interaction between paging and normal file system traffic, we created a special file system to be used only for paging. This file system was only used for paging and never used to hold normal files, and conversely the normal file system was never used for paging during the execution of the benchmarks.

This special file system was created in the disk partition used by HP-UX 2.0 for swap space.

- On HP-UX 2.0 the magic number of an executable file can specify that the program be preloaded or loaded on demand. Tut kernels ignore this feature, loading all programs on demand regardless of what the magic number specifies. HP-UX 2.0 does obey the magic number and preloads a text segment if that is specified. Since nearly all executables have a magic number that specifies preloading, this policy difference between Tut and HP-UX 2.0 may have an effect on their relative performance.
- The MUB and Paging benchmarks kept their files on dedicated discs to minimize I/O congestion.

## 7 Benchmark results

This section presents the results of the benchmarks in three parts. First, in Section 7.1 we present comparisons of the performance of Tut kernels with experiments enabled to the base Tut kernel for the Andrew and MUB benchmarks. Next in Section 7.2 we present similar results for the Lazy Mapout and the Paging benchmarks. Finally in Section 7.3 we compare the performance of Tut kernels to that of HP-UX 2.0 for the Andrew, MUB, and Lazy Mapout benchmarks.

The results are presented in graphs using labels to indicate the type of Tut kernel on which the corresponding measurement was taken. A label is a string of four characters. The first character is either "R" or "\_". "R" indicates that the Tut kernel used read-only aliasing. "\_" indicates that it did not. Similarly, "P" in the second position indicates that `pmap_pre_enter()` was used, "L" in the third position indicates that lazy mapout was used, and "S" in the fourth position indicates that shared text was used. For example, the label "RP\_S" indicates a Tut kernel that has read-only aliasing, `pmap_pre_enter()`, and shared text enabled, and the label "\_\_\_\_" indicates the base Tut kernel that has none of the experiments enabled.

We ran each benchmark enough times so that the standard deviations of the quantities shown in the graphs are less than or equal to 0.01.

## 7.1 Andrew and MUB on Tut

We first look at the relative performance of Tut kernels with some combination of the experiments enabled to the base Tut kernel for the Andrew and MUB benchmarks. Figure 2 (at the end of this paper) presents the results for the Andrew benchmark. It displays the ratios of the mean elapsed time to complete the Andrew benchmark on a given Tut kernel to the mean elapsed time to complete the Andrew benchmark on the base Tut kernel using the same page size. For example, the `R_` Tut kernel (i.e. a Tut kernel with read-only aliasing enabled) using a 2K page size has an elapsed time that is 0.91 that of the base Tut kernel using a 2K page size. Please note that in all of these figures, smaller numbers are "better." Figure 3 presents the corresponding results for the MUB benchmark. From these figures we see that enabling shared text and/or read-only aliasing provides a substantial performance improvement over the base Tut kernel. The Tut kernels that perform the best are the `_P_S`, `RP_S`, and `RPLS` kernels.

A comparison of the results with shared text enabled to similar kernels without shared text shows that any Tut kernel that was not using read-only aliasing decreased the elapsed time for the Andrew and MUB benchmarks by approximately 8%. Adding shared text to a Tut kernel that was already using read-only aliasing still decreased the elapsed time, but then only by approximately 3%.

Looking at the results for adding read-only aliasing to other combinations of kernels is quite similar to adding shared text, but with their roles reversed. Adding read-only aliasing to a kernel that was not using shared text decreased the elapsed running time by 8%. Adding it to a kernel that was using shared text decreased the elapsed time by approximately 2%.

Adding `pmmap_pre_enter()` to any Tut kernel always decreased the mean elapsed run time by 2 or 3%. This shows that `pmmap_pre_enter()` was very well behaved; adding it to any Tut kernel had nearly the same positive effect.

Using lazy mapout in a Tut kernel provided little or no benefit when running the Andrew and MUB benchmarks. This is not what we expected. We thought that adding lazy mapout to a kernel using shared text would be a clear win. It was this result that led us to run the Paging and Lazy Mapout benchmarks. These benchmarks were designed to test the specific operations in which lazy mapout should be a performance win. The results of those benchmarks are discussed next.

## 7.2 Paging and Lazy Mapout on Tut

In this section we look at the results of two benchmarks that were used to make additional measurements of the performance effects of adding lazy mapout to a Tut kernel.

When we ran the Paging benchmark, the total elapsed times were virtually identical for all of the kernels with identical page sizes which indicates that the Paging benchmark was I/O bound. The mean elapsed time to run the Paging benchmark on a Tut kernel using an 8K page was about 3/4 that of a Tut kernel using a 4K page size. This was not surprising since a 4K Tut kernel must do twice as many paging operations as an 8K kernel during the benchmark.

However, enabling experiments in a Tut kernel does have an effect on the mean *system* times for the Paging benchmark as shown in Figure 4. This figure shows the ratio of the mean system time for a Tut kernel to the mean system time on the base Tut kernel using the same page size. Here we see that the experiments did decrease the system time.

Figure 5 shows the ratios of the mean elapsed time of the Lazy Mapout benchmark on a given Tut kernel to the mean elapsed time of the Lazy Mapout benchmark on the base Tut kernel using the same page size. Figure 6 shows the ratios of the mean system time of the Lazy Mapout benchmark for a given Tut kernel to the mean system time on the base Tut kernel using the same page size on the Lazy Mapout benchmark.

Comparing the system times for kernels with lazy mapout to similar kernels without this option shows that for these benchmarks lazy mapout did provide a performance improvement for nearly every Tut kernel.

### 7.3 Tut and HP-UX 2.0

We end with a comparison of the performance of the Tut kernels to HP-UX 2.0 when running the Andrew, MUB, and Lazy Mapout benchmarks. Figure 7 presents the ratio of the mean elapsed time for the Andrew benchmark on each Tut kernel to the mean elapsed time on HP-UX 2.0. This figure presents the results for Tut kernels using virtual page sizes of 2K, 4K, and 8K. In all cases the mean time of the Andrew benchmark on HP-UX 2.0 is for a virtual page size of 2K, since this is the only page size allowed in HP-UX 2.0. Figure 8 presents the corresponding results for the MUB benchmark. Figure 9 shows the ratios of the elapsed time of the Tut kernels to the elapsed time of the Lazy Mapout benchmark on HP-UX 2.0 using a page size of 2K. The numbers reported in all three figures have a standard deviation of at most 0.01.

We see that the best performing Tut kernel were slightly slower than HP-UX 2.0 on the Andrew benchmark and that the best performing Tut kernels were slightly faster than HP-UX 2.0 on the MUB and Lazy Mapout benchmarks.

## 8 Conclusions

We now present our conclusions.

- Shared text, read-only aliasing, and `pmap_pre_enter()` definitely provided performance improvements over the base Tut kernel. The performance benefit of lazy mapout was not always obvious but was apparent on benchmarks that specifically measured the areas in which it should have helped. Lazy mapout did not degrade system performance on the benchmarks on which it provided no performance gain and so it too should be considered a small win.
- It was worth making the minor modifications to the Mach machine independent code that were needed to implement our experiments. They provided good performance improvements on our architecture (8-10% reduction in elapsed time when using shared text and 2-3% when using `pmap_pre_enter()`) and should have no effect on the performance of Mach on machine architectures that do not need them.
- Replacing the HP-UX 2.0's virtual memory and process management subsystems with Mach 2.0's was essentially a performance no-op when enough of the experiments were turned on in the Tut kernel.
- We had to work harder on our virtually addressed cache machine architecture to regain performance advantages that physically addressed cache architectures get for free when running Mach. Mach's assumption that virtual address aliasing is a cheap, efficient way to share memory between tasks is true on physically addressed cache machine architectures but is not necessarily true on machines with a virtually addressed cache.

## 9 Related work

Sharing writable memory between tasks on a virtually addressed cache architecture can only be done by remapping unless some other special techniques are used. One such technique is to map shared memory in each task's address space so that all virtual aliases reference the same line in the cache. For example, a virtually addressed cache could be constructed so that virtual addresses that differ by  $2^{20}$  map to the same cache line. The kernel could take advantage of this property by mapping shared writable memory to virtual address ranges that differ by multiples of  $2^{20}$  eliminating the need to remap the shared physical pages as they are accessed. This technique may put restrictions on where user tasks can map memory. Currently, in Mach the only restriction is that a task map memory on a page boundary. A disadvantage of this technique is that it limits the size of the cache to at most  $2^{20}$  bytes. Another technique is to just take advantage of PA-RISC's global virtual address space. The HP-UX 2.0 implementation of System V shared memory used this approach. All users of a System V shared memory segment access it using the same range of virtual addresses.

## 10 Acknowledgments

We would like to thank David Black, who provided lots of help and insight during our port of Mach to PA-RISC and HP-UX; Ahmed Ezzat, who got the Tut project started and who worked with us during the first phase of the project; Jim Hays, who proposed many of the experiments described here; David Jacobson, who helped us make the graphs; the Unix Kernel Lab (UKL), who provided us with much of the code needed to implement read-only aliasing and the Andrew and MUB benchmark programs; and John Wilkes, who provided us with management and encouragement.

## References

- [Clegg 86] Frederick W. Clegg, Gary S. Ho, Steven R. Kusmer, and John, R. Sontag, *The HP-UX Operating System on HP Precision Architecture Computers*, Hewlett-Packard Journal, (December 1986).
- [HP 87] Hewlett-Packard, *Symbolic Driver Debugger User's Guide*, Manual Part No. 91834-90005, (October 1987).
- [HP 89] Hewlett-Packard, *Precision Architecture and Instruction reference manual*, (P/N 09740-90014), Edition 3, Hewlett-Packard, (April 1989).
- [Lee 89] Ruby B. Lee, *Precision Architecture*, Computer, Vol. 22, No. 1, (January 1989).
- [Mahon 86] Michael J. Mahon, Ruby Bei-Loh Lee, Terrence C. Miller, Jerome C. Huck, and William R. Bryg, *Hewlett-Packard Precision Architecture: the processor*, Hewlett-Packard Journal, (August 1986).
- [Tevanian 87a] Avadis Tevanian, *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach*, Ph.D. dissertation and technical report (CMU-CS-88-106), Department of Computer Science, Carnegie Mellon University, (December 1987).
- [Tevanian 87b] A. Tevanian and R. F. Rashid, *MACH: A Basis for Future UNIX Development*, Technical report (CMU-CS-87-139), Carnegie-Mellon University, (June 1987).



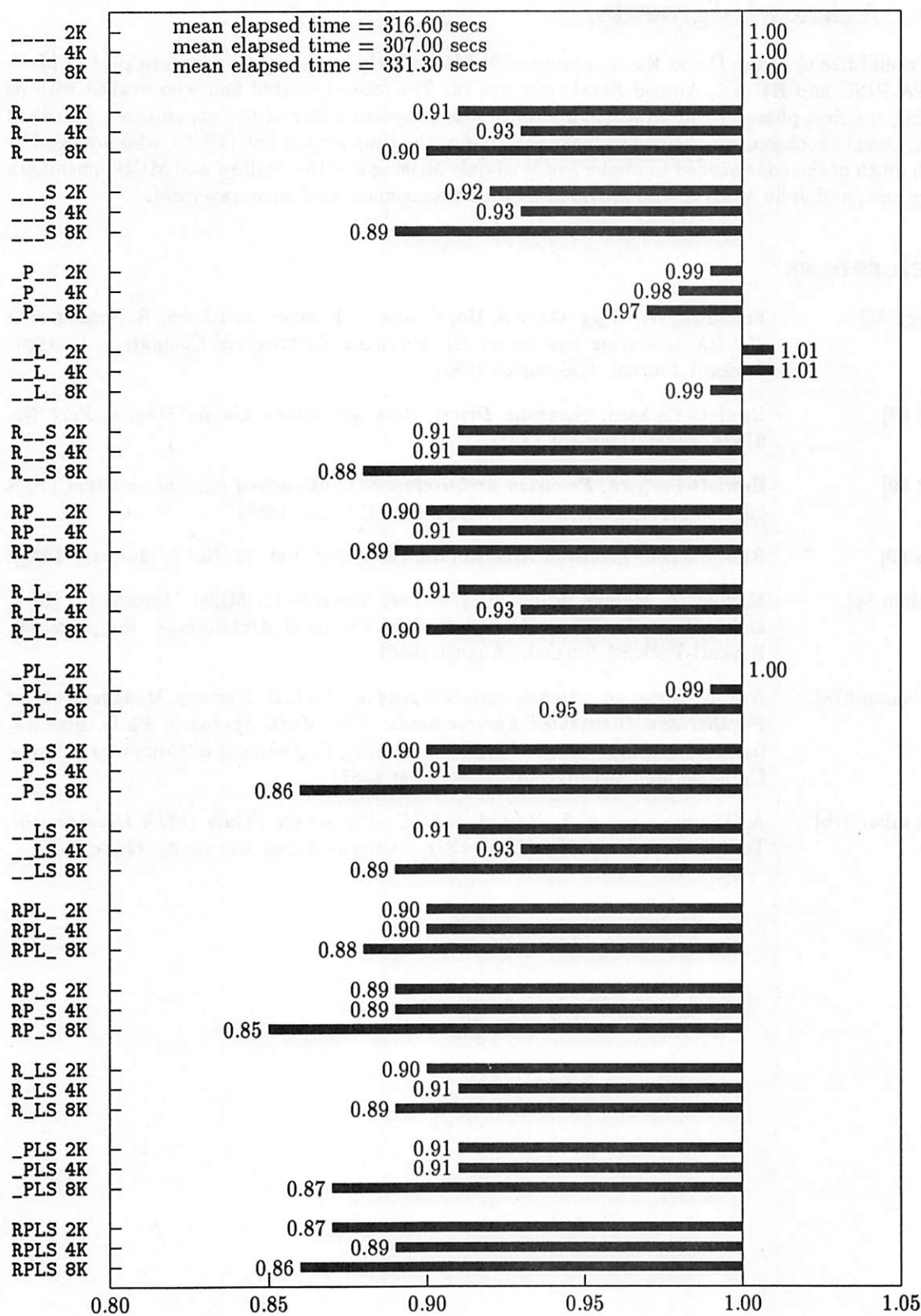


Figure 2: Ratio of elapsed times for Andrew (2 users) on Tut kernels to elapsed time of Andrew on a base Tut kernel using the same page size. (Please note that smaller numbers are "better.")



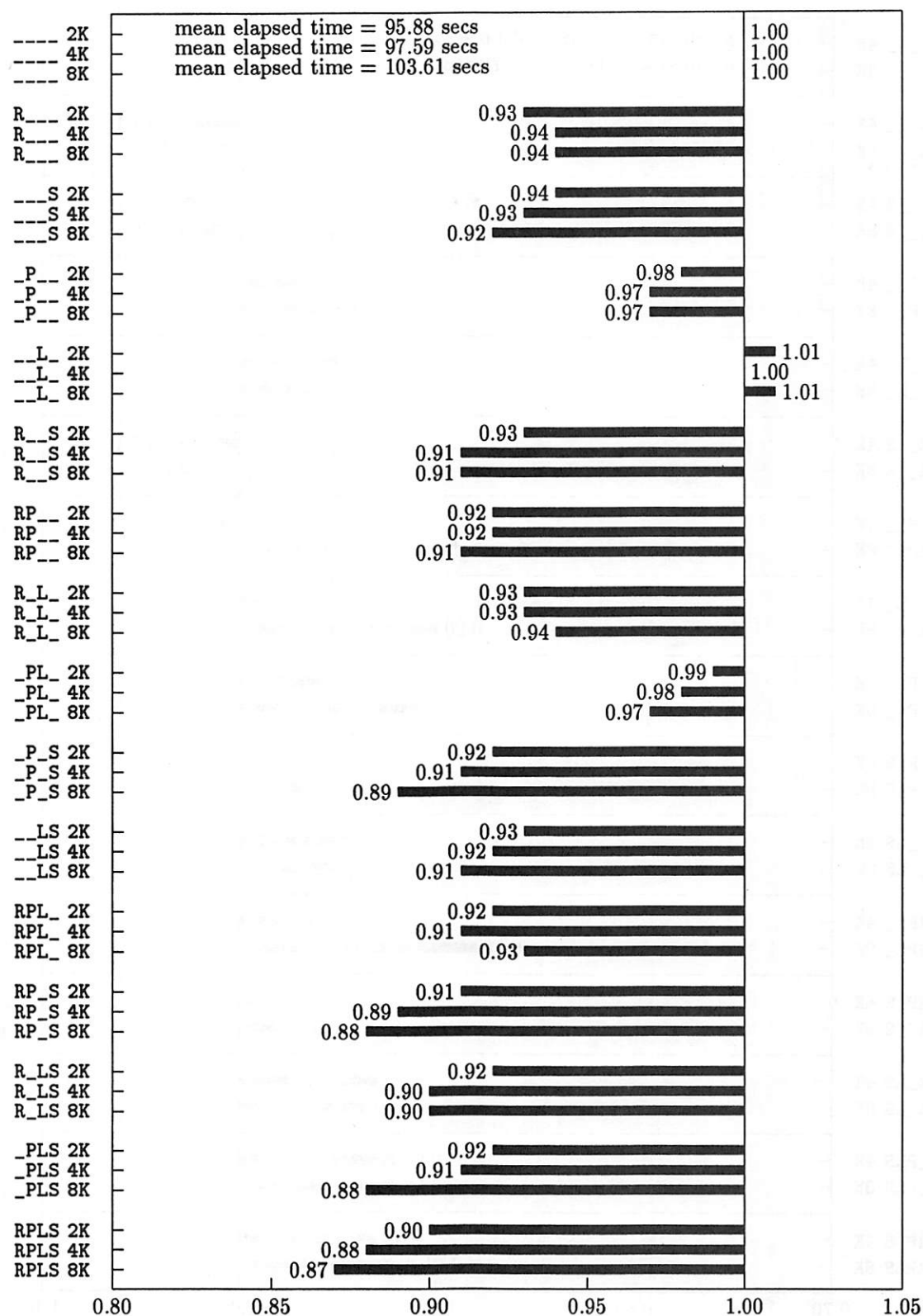


Figure 3: Ratio of elapsed times for MUB (3 users) on Tut kernels to elapsed time of MUB on a standard Tut kernel using the same page size.

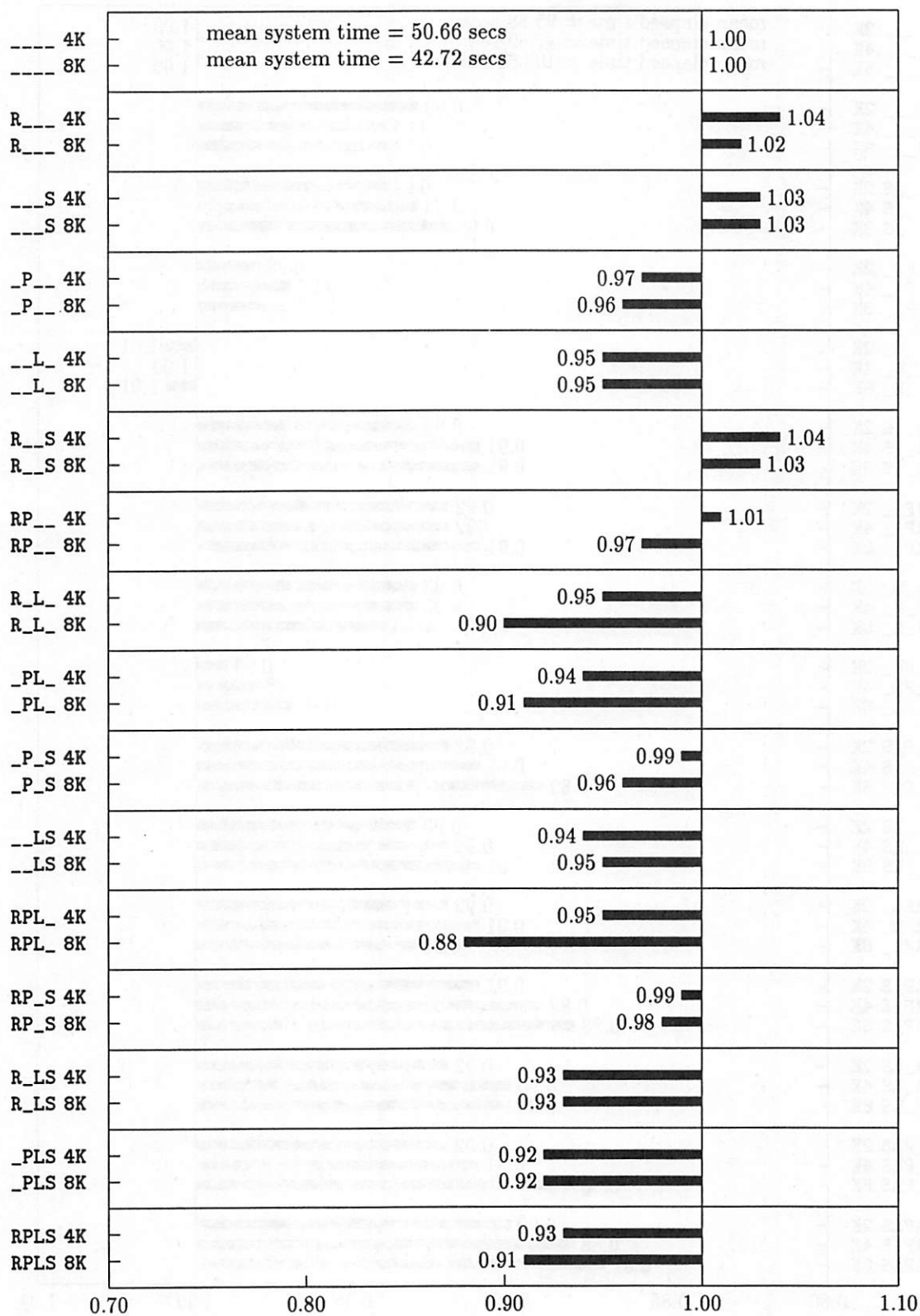


Figure 4: Ratio of system times for the Paging benchmark on Tut kernels to system time of the Paging benchmark on a base Tut kernel using the same page size.

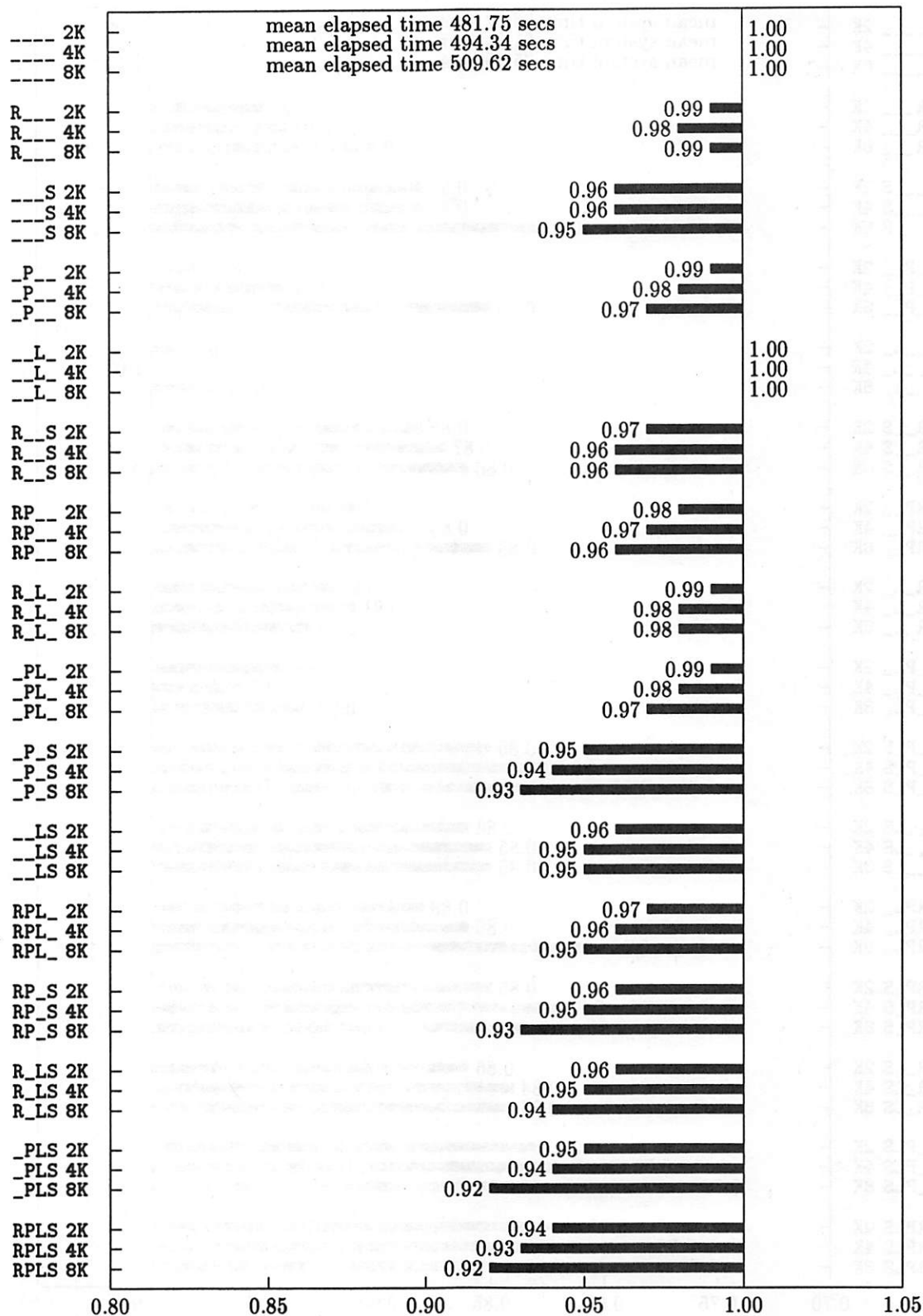


Figure 5: Ratio of elapsed times for the Lazy Mapout benchmark on Tut kernels to elapsed time of the Lazy Mapout benchmark on a base Tut kernel using the same page size.

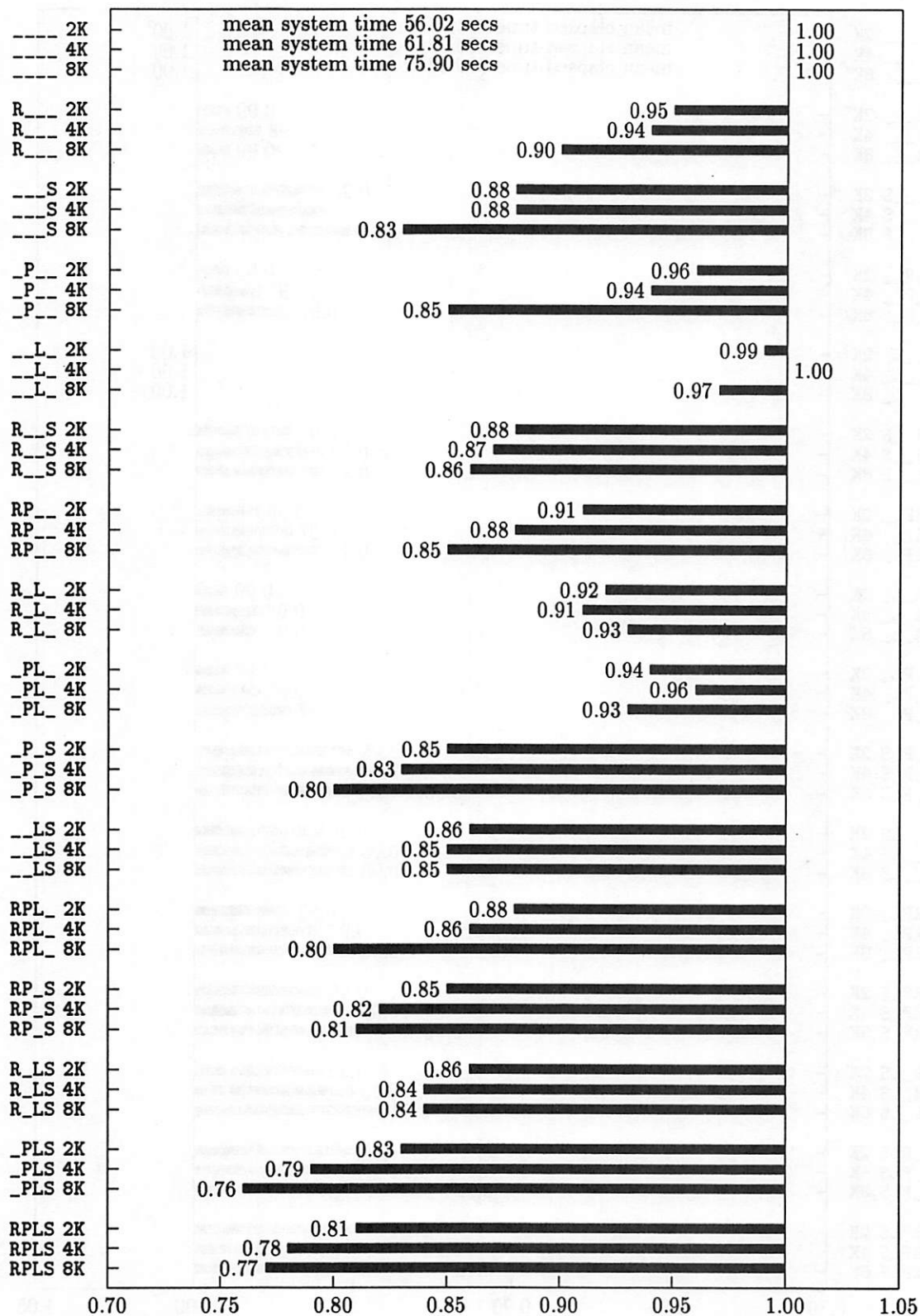


Figure 6: Ratio of system times for the Lazy Mapout benchmark on Tut kernels to system time of the Lazy Mapout benchmark on a base Tut kernel using the same page size.

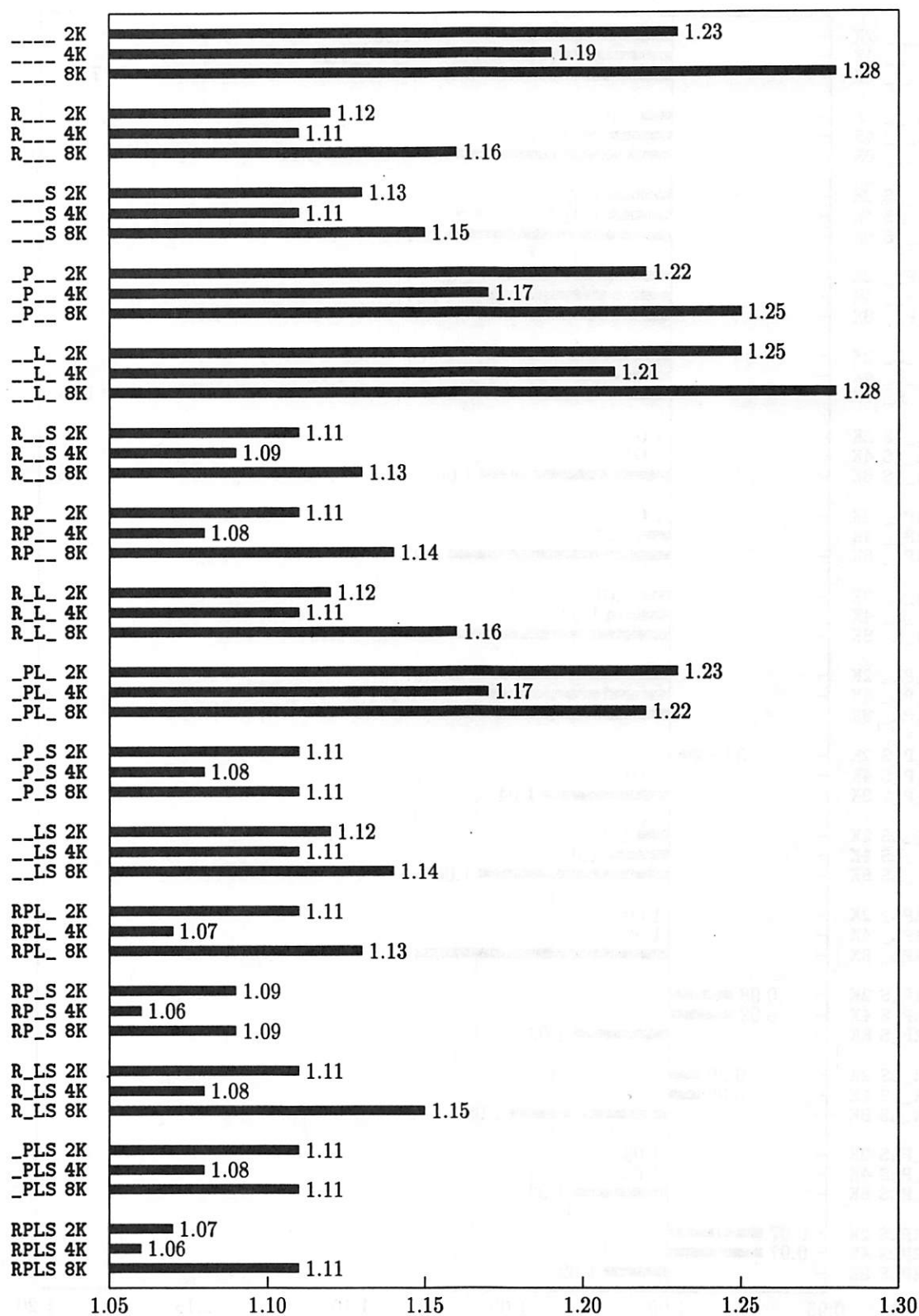


Figure 7: Ratio of elapsed times for Andrew (2 users) on Tut kernels to elapsed time of Andrew on HP-UX 2.0.



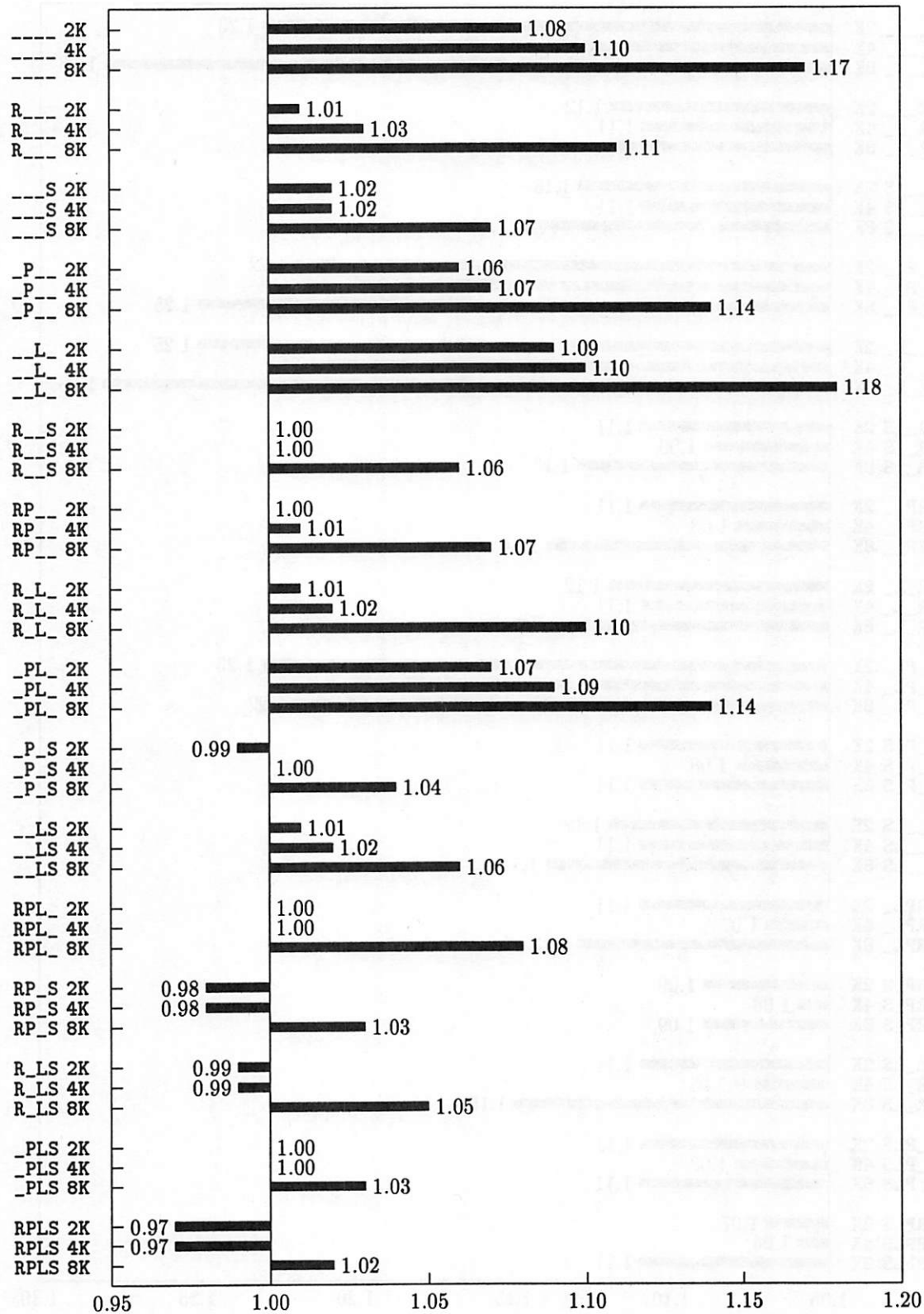


Figure 8: Ratio of elapsed times for MUB (3 users) on Tut kernels to elapsed time of MUB on HP-UX 2.0.

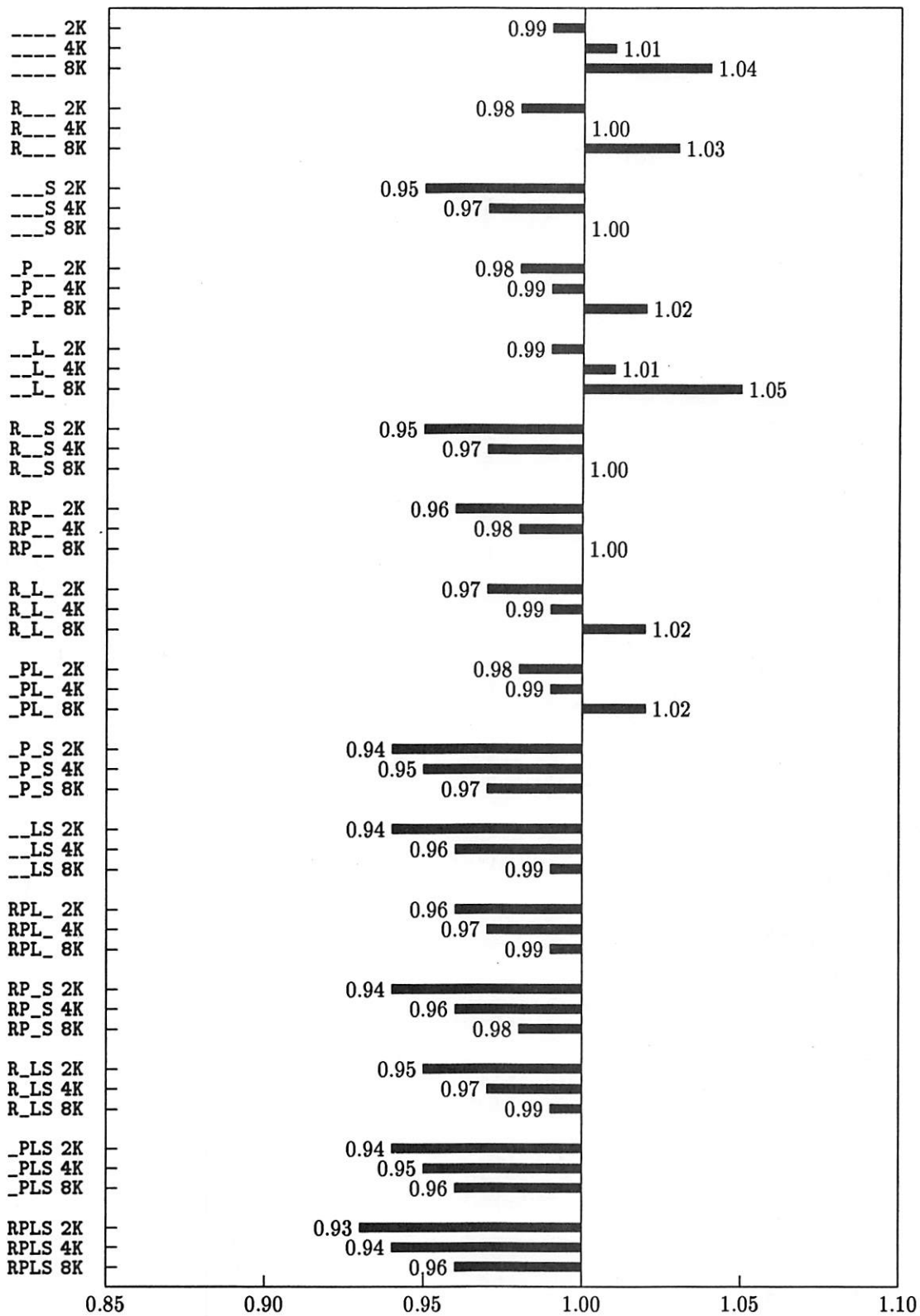


Figure 9: Ratio of elapsed times for the Lazy Mapout benchmark on Tut kernels to elapsed time of the Lazy Mapout benchmark on HP-UX 2.0.



# The Mach Timing Facility: An Implementation of Accurate Low-Overhead Usage Timing

David L. Black\*  
School of Computer Science  
Carnegie Mellon University  
David.Black@cs.cmu.edu

## 1 Introduction

This paper describes new techniques for accurate measurement of processor usage and their implementation in the Mach operating system. Current statistical methods for measuring processor usage can be inaccurate and may yield distorted results. Directly measuring the usage is more complicated and may impose unacceptable overheads. The Mach timing facility demonstrates that this is not the case; it is an implementation of accurate usage timing via direct measurement (using timestamps) that exhibits low overheads. A minimal level of hardware support is required to provide a timestamp source. The facility's design uses techniques developed by Lamport to allow concurrent data structure access without multiprocessor interlocks (e.g., test-and-set instructions). This permits the use of timing code in interrupt service routines without adding uncertainty to interrupt service times. The design and implementation of the timing facility are described in this paper, and performance measurements are presented that demonstrate its low overhead cost. An appendix provides a detailed description of the in-kernel interface to the timing facility and guidelines for porting it.

---

\*This research was supported by the Defense Advanced Research Projects Agency (DOD) and monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251, ARPA Order No. 5993. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. government.

The common statistical methods of measuring processor usage are imprecise and can yield inaccurate results. These methods are based on periodic clock interrupts; at each interrupt, the interrupted activity is charged as if it had used the entire preceding period. In a statistical sense, each interrupt samples the usage of the processor; over longer periods of time the distribution of the samples will match the distribution of usage in the absence of anomalies. Sampled measurements are imprecise when the number of samples is small (i.e., for measuring short periods of time), and there are several anomalies that affect results obtained from these samples. These include problems in assigning interrupt overheads, processing activities synchronized to the clock interrupts, and cyclic behavior in applications.

Direct measurement of time is an alternative to sampled measurements, but requires hardware support and more implementation effort. Direct measurement is based on sampling a time source each time the usage of the processor changes and charging the resulting time periods to each use. It requires a hardware source of timestamps to measure these time periods, and software management to accumulate the usage times. These measurements avoid the distortions caused by sampling anomalies and can be several orders of magnitude more precise. The actual precision achieved depends on the timestamp source and the software overheads. Results from the Multimax implementation of Mach's timing facility show that a system based on a microsecond timestamp source can achieve three orders of magnitude improvement in accuracy. The need to obtain timestamps and update usage times on a frequent basis introduces additional overhead, but its impact is minimal. This paper also recommends hardware designs to provide support for accurate usage timing.

This paper describes the design and implementation of a timestamp-based direct measurement usage timing facility for Mach, and demonstrates that its overheads and required hardware support are minimal. Section 2 covers timing methodologies, including the problems caused by statistical timing and the alternative of using timestamp-based timing. Section 3 describes the design of the timing facility used in Mach. The actual implementation for the Encore Multimax is described in Section 4, and Section 5 presents performance measurements from that implementation to demonstrate its low overheads. An appendix describes the internal kernel interface to the timing facility and offers some guidelines and hints for porting it.



## 2 Usage Timing Techniques

Statistical timing samples the use of each processor and assumes that the samples are representative of actual use. This is the most common technique for measuring processor usage, and is usually implemented by a clock that generates interrupts at a known rate (100Hz is often used). Code in the interrupt handler determines what was interrupted (e.g., which thread, user or kernel mode) and charges it with one tick of usage (the period since the last interrupt, 10ms for a 100Hz clock). In the statistical sense, the use of the processor is sampled by each clock interrupt; over reasonable periods of time the samples should provide a good approximation to actual usage in the absence of sampling anomalies. This is adequate for scheduling purposes. Improvements could be obtained from more accurate information, but a large number of successful timesharing systems are based on this technique.

There are a number of drawbacks to statistical timing. It is designed for measuring long term averages, so its precision for short term measurements is limited by the clock interrupt rate. Many systems have been reducing this clock interrupt rate to reduce overhead. These reductions cause a loss of precision in timing measurements and lengthen the period over which they must be averaged to attain a given degree of accuracy. These measurements are of no use in measuring times that are on the order of a few seconds or less, but such measurements are of great importance in a multiprocessor because it may be difficult or impossible to reliably reproduce short-term synchronization behavior that has a major impact on performance. Statistical timing also suffers from a distortion problem in that the overhead of the clock interrupt routine and any activities synchronized to it (e.g., network protocol timeout processing) are invisible. Such activities usually complete before the next clock interrupt and are therefore never interrupted and sampled. As a result, the time consumed by such activities is hidden in the time charged to other uses of the processor, resulting in a consistent overestimation of usage times by several percent. This effect can be aggravated by usage patterns that happen to synchronize with the clock interrupt frequency; in the worst cases, the resulting usage measurements may bear no resemblance to the actual usage pattern [5]. For an extreme example reported by Wendorf [5] the usage times measured for a specially constructed pair of programs differed from the actual times by a factors of more than 20.

Timestamp-based timing tracks what each processor executes and accumulates time measurements accordingly. This increases both the the preci-

sion and accuracy of the measurements, in part by avoiding the distortions of statistical timing. Timestamp-based timing accumulates all time on individual timers. There are user and kernel mode timers for each thread in the system, and each processor in the system has at least one timer for execution that does not occur in a thread context (primarily interrupt service routines); more than one of these *kernel timers* can be used to distinguish among these activities (e.g., different interrupts). Operating system code is responsible for switching among timers to make sure that the correct timer is accumulating time for each processor. To allocate processor time among the various timers, there are five places where such switches must be made:

1. In response to a trap, fault, or system call from user mode: switch to the system timer for the thread.
2. As part of returning to user mode from a trap, fault, or system call: switch to the user timer for the thread.
3. In response to an interrupt: switch to the kernel timer for the interrupt.
4. As part of returning from an interrupt: switch back to the timer that was running before the interrupt.
5. As part of a context switch: switch to the system timer for the new thread.

These switches cause user timers to accumulate user-mode execution time, system timers to accumulate system-mode execution time, and kernel timers to accumulate kernel overhead time. Time spent servicing interrupts is considered kernel overhead to avoid charging it to threads not involved in generating the interrupts. Idle time is accumulated by the system timers for idle threads.

There are several problems that have limited the use of timestamp-based timing. The most serious problem is that many hardware implementations do not provide a source of timestamps. This is a simple omission that should be corrected in the hardware, as it is relatively cheap and easy to provide a counter that runs at some known clock rate. This counter should be provided in a page of memory that can be mapped into applications to allow them to efficiently obtain timestamps<sup>1</sup>. Even if the appropriate hardware support is present, the increased accuracy of the resulting measurements

---

<sup>1</sup>Mach's *device pager* can be used to implement this mapping.

makes them more complicated to manage. Statistical usage timing can use a single word of memory to hold the accumulated tick count, and can rely on memory subsystem hardware to synchronize accesses from multiple processors. The additional precision present in timestamp-based measurements requires that more than one word of memory be used to hold accumulated time, and therefore requires software protocols to manage concurrent accesses to this memory. This problem is compounded by the fact that code to update timestamp-based times occurs in critical paths of the kernel such as trap, interrupt, and system call processing. This makes performance a paramount concern, and the use of mutual exclusion locking (e.g., test and set) undesirable.

### 3 The Mach Timing Facility

The design of Mach's timestamp-based timing facility involves one data structure with four operations, the *timer*. This can be thought of as a stopwatch; at any time it is either running and accumulating time or stopped. Restarting continues the accumulation from the point at which it stopped. Each thread has a timer for user and system time, and in addition the kernel has one or more timers for interrupt-related activities. At any time there is exactly one timer running for each processor in the system; a user timer if the processor is executing a thread in user mode, a system timer if the processor is executing a thread in kernel mode, or a kernel timer if the processor is executing an interrupt handler. Idle processors accumulate time on the system timer for their idle thread. A timer consists of two words of memory to hold the accumulated time (*low\_bits*, *high\_bits*), a check field used to avoid access conflicts (*high\_bits\_check*), and a timestamp field to hold the timestamp that started the timer running (*tstamp*). Two words of memory are needed to hold the accumulated time in a timer because one word overflows too quickly. For a common case of microsecond timestamps and 32-bit words, the overflow period for one word is about 70 minutes. This structure is shown in Figure 1.

There are four basic operations on timers:

1. **Start.** Start a timer running.
2. **Update.** Stop a timer and update the time it contains.
3. **Normalize.** Normalize the accumulated time values.

low_bits
high_bits
high_bits_check
tstamp

Figure 1: Timer structure

#### 4. **Read.** Read the value in a timer.

The **Start** operation is used during the boot sequence to start the initial timer for each processor. Thereafter each trap, trap return, interrupt, interrupt return, and context switch performs an **Update** operation on the current timer and a **Start** operation on a new one using the same timestamp. This combination of **Update** and **Start** with the same timestamp is called **Switch**. Traps and trap returns that do not change modes (i.e., go back to system mode) do not change the running timer. Interrupts must be disabled when switching between timers to make the **Switch** operations atomic; this maintains the invariant of one timer running at any time on each processor. The **Normalize** operation moves time within the timer between fields denominated in different time units (e.g., seconds, microseconds) to avoid overflow. It is invoked as needed by **Switch** operations. **Read** operations are used in user-invoked system calls to read the value of a timer. The scheduler uses an optimized version of **Read**, called **Delta** to maintain the usage information that drives the scheduling algorithm.

Times are stored in a non-normalized format using units appropriate to the timestamp source to make the **Switch** operation as fast as possible. The non-normalized format permits the low order word of accumulated time to hold more than one unit of the high order word. For example, if the units for the low and high order words are microseconds and seconds respectively, then the microsecond word may hold more than one million microseconds. The units for the low order word are chosen to match the timestamp source, so an **Update** involves a subtraction to calculate the elapsed time and an addition to add it to the low order word accumulated time field; no conversion or normalization operations are required. Units are converted if needed (e.g., if the timestamps come from an 8MHz clock, division by 8 is necessary) by **Read** operations because these operations are less frequent and do not occur in critical paths. The special **Delta** operation used by

the scheduler does not do unit conversion; instead, the scheduler keeps its internal statistics in the same units as the hardware timestamps. Conversion from these machine dependent units occurs in the scheduler's priority update calculations via shift and add operations. Up to two shifts and an add can be configured to implement any desired conversion factor. Normalization overhead is reduced by delaying normalization until the low order word of the accumulated time exceeds  $2^{31}$  (for 32-bit words); this is about 35 minutes of accumulated time for microsecond timestamps. This overflow is checked by each **Update** operation, except that an **Update** caused by an interrupt defers this overflow check until the corresponding interrupt return to improve interrupt response.

Conflicting write accesses to timers are eliminated by restricting write operations (**Switch** and **Normalize**) to a single writer for each timer. For a usage timer associated with a thread, the writer is that thread. For kernel timers, this is achieved by associating each timer with a specific processor. As a result, kernel timers are grouped in arrays with one timer for each processor on a multiprocessor. Currently, a single set of kernel overhead timers is used, but different sets could be associated with different interrupts. This would yield information on how much time is spent servicing each interrupt source and how that time is distributed among the processors.

Read/write conflicts are simplified by defining the **Read** operation so that most conflicts are resolved by hardware. The **Read** operation is defined to only return time from the accumulated time fields, ignoring time since the last **Start** operation for a running timer. The clock interrupt rate limits the resulting inaccuracy; for a 100Hz clock rate, no more than 10 milliseconds will elapse before the next event that updates a running timer (context switch, trap, trap return, interrupt, interrupt return). More accurate values can be obtained by stopping the timer (e.g., suspend the corresponding thread) or causing an event to occur (e.g., interprocessor interrupt) if this is important. This definition of the **Read** operation makes the timestamp field of the timer private to the **Start** operation, removing it from participation in read/write conflicts. The vast majority of read/write conflicts are with an **Update** operation on the write side, and hence only involve the low order accumulated time field. The memory hardware resolves these conflicts by ordering accesses to this field. In both cases the **Read** operation returns valid data because the order of the accesses determines whether the **Read** appears to have occurred before or after the **Update**.

Conflicts with the **Normalize** operation are resolved by a backout protocol. Because the **Normalize** operation writes both accumulated time fields,



it is possible for a naive **Read** operation to obtain one pre-normalization value and one post-normalization value. This is avoided by using the check field to implement a backout protocol that combines features of Lamport's Two-Digit monotonic clock algorithm[4] and his solution to the single writer case of the readers/writers problem[3]. The check field holds a copy of the high order accumulated time field of the timer. **Normalize** operations write the normalized accumulated values in the order check-low-high. **Read** operations read the fields in the reverse order, high-low-check, and test to see if the high and check fields match. No **Normalize** operation could have been in progress if they match<sup>2</sup>, so the data read from the timer is valid. If the high and check values don't match, the values read from the timer have been corrupted by the **Normalize** (the low value goes with one of the two high values, but it's impossible to tell which one), so the **Read** operation is retried. This retry always succeeds because **Normalize** operations are infrequent.

Figure 2 shows pseudo-code for the major timer operations. For simplicity, the timestamp units are assumed to be microseconds (the base unit of the **Read** operation); if this is not the case, the **Read** operation would contain additional conversion code to change units. This example keeps the `high_bits` field of the timer in seconds for convenience; **Normalize** operations are infrequent enough that choosing the unit for the convenience of **Read** makes sense. The actual interface used within the kernel to access and manipulate the timers is documented in the Appendix.

The scheduler's **Delta** operation (optimized **Read**) uses a copy or *snapshot* of the timer to return elapsed time. The operation compares this snapshot to the current timer value, returns the elapsed time, and updates the snapshot. An efficient implementation is obtained by observing that the high order field of the snapshot will usually match the timer; only the low order fields of the timer and snapshot are involved in most instances. The implementation reads the low field, and then compares the high field of the snapshot with the check field of the timer. This is the high-low-check order of accesses used by a **Read** operation except that the high value comes from the last snapshot instead of the current operation. If the high values match, then the difference in low values is the elapsed time and the new snapshot is obtained from the old by updating the low field. If the high values don't

<sup>2</sup>This assumes that performance of write operations cannot be reordered by hardware. For hardware that violates this assumption (e.g., PLUS[1]), fence operations must be inserted between the writes performed by the **Normalize**.

```

#define TIMER_HIGH_UNIT 1000000

Start(timer, timestamp)
{
    timer->tstamp = timestamp; }

Update(timer, timestamp)
{
    timer->low_bits += timestamp - timer->tstamp; }

Normalize(timer)
{
    unsigned    high_increment;

    high_increment = timer->low_bits / TIMER_HIGH_UNIT;
    timer->high_bits_check += high_increment;
    timer->low_bits %= TIMER_HIGH_UNIT;
    timer->high_bits += high_increment;
}

Read(timer, seconds, microseconds)
{
    unsigned save_high, save_low;

    do {
        save_high = timer->high_bits;
        save_low = timer->low_bits
    } while (save_high != timer->high_bits_check);

    microseconds = save_low % TIMER_HIGH_UNIT;
    seconds = save_high + (save_low / TIMER_HIGH_UNIT);
}

```

Figure 2: Timer operation pseudo-code

match, a **Read** operation is performed to obtain a new snapshot from which the elapsed time is then calculated. This **Read** operation is overkill in the current system because **Delta** operations are only executed by the thread on its own timers or by some other thread when the target is known not to be running; in both cases, **Normalize** operations are implicitly excluded. The performance impact of these **Read** operations is minimal because they only occur after **Normalize** operations, which are rare. The resulting implementation is extensible to schedulers that do need to read elapsed times from running threads.

The algorithms used in Mach's timing facility are based on algorithms developed by Lamport, but there are several differences and extensions. Lamport's two-digit monotonic clock algorithm[4] maintains a clock in normalized format, and uses this to allow the reader to infer a valid clock value if a read and a write should overlap. Mach's timing facility deliberately uses a non-normalized format to favor writes over reads by limiting the effect of write operations. The cost of not using a normalized format is that a read operation cannot infer a valid timer value if it detects an overlapping write (which must be a **Normalize**). Instead, the read must be retried, as per Lamport's general readers/writers solution[3]. The rarity of **Normalize** operations in this case guarantees that the retry will succeed. In addition, the application of this technology to usage timing is new; the motivating application for Lamport's work was the difficulties encountered in sharing multiple word time of day clocks between applications and the operating system. The use of a lock that can be held by applications or the kernel is clearly unacceptable because it puts the kernel at the mercy of a malicious application. Work is in progress to use Lamport's monotonic clock algorithm to implement this feature (time of day clock in shared memory) for Mach.

## 4 Implementation

There are several options for timestamp sources. The Multimax implementation uses a shared free running 32-bit microsecond counter as the timestamp source. Reading this counter takes 2.5 microseconds because it bypasses the processor caches and accesses special hardware on the System Control Card (this is slower than a memory access caused by a cache miss). All accesses to this counter are serialized by the bus protocol, so it is a potential source of multiprocessor contention. An alternative hardware design is to provide

a timestamp source for each processor. Since timestamps are only compared to others from the same processor, it is not necessary to synchronize these sources. In other words, implementing a distributed timestamp source for Mach's timing facility does not require synchronized clocks. Counters that roll over at values other than  $2^{32}$  can be used; the rollover period must be long enough to be detected reliably. The detection mechanism used notices that the difference between two timestamps is negative, infers that exactly one rollover has occurred, and compensates by adding the rollover period. A useful practical criterion for reliable detection is that each processor's counter sampling rate (for timestamps) should be at least twice the counter's rollover rate. A periodic countdown counter that generates an interrupt on rollover is not sufficient without an additional rollover detection mechanism (e.g., a bit set by hardware when the rollover occurs). In the absence of such a mechanism, delays in accepting clock interrupts may cause the timing facility to miss rollovers. This happens when a rollover interrupt is delayed to the extent that its interrupt entry timestamp is greater than the interrupt exit timestamp of the previous interrupt's service routine. Generating an additional interrupt at the midpoint of the timer's rollover period is another way to avoid this problem.

The Multimax implementation uses inline assembly language for the timing operations in critical paths. This uses approximately one third fewer instructions than a previous implementation based on C routines. The major savings in instruction count for the assembly language version come from avoiding the instructions that implement a procedure call (three on entry, three on exit). An additional savings is realized in the trap cases by optimizing the protocol used to disable interrupts. The Multimax enables and disables interrupts via an off-chip ICU, but a chip bug requires that the processor internally disable interrupts before manipulating the ICU and re-enable them afterwards. This results in three instructions for each enable and disable, or a total of six instructions per timing operation. The timing code for traps reduces this to two instructions per operation by disabling interrupts on the processor for the duration of each operation. This technique must be used with great care because non-maskable interrupts are also routed through the ICU, and are therefore disabled by this technique.

## 5 Performance

Timestamp-based timing can achieve orders of magnitude improvements in timing accuracy. On a Multimax with a 10Hz clock interrupt rate, statistical timing is of little use for periods of less than several seconds; at one second, the intrinsic measurement error is 10%, and this drops to 2% at five seconds. In contrast, Mach's timing facility can time periods of less than one hundred microseconds on the same hardware with excellent accuracy and repeatability. This represents an improvement more than four orders of magnitude ( $10^4$ ) in the precision of the timing facility. For a more common 100Hz clock rate, the corresponding improvement with a microsecond timer is in excess of three orders of magnitude. Additional improvements in accuracy are obtained by correctly excluding kernel overhead from user timing measurements. These improvements in precision and accuracy have delivered valuable new timing capabilities to users at CMU and elsewhere.

Timing overhead measurements were taken from the Multimax implementation for the **Switch** operations in the critical paths of the kernel. Because these measurements rely on the shared counter to compute elapsed times, every critical path accesses the counter three times during these tests, causing the results to overstate the effects of access contention for the counter. Each operation was measured 2048 times on a Multimax with ns32332 processors (about 2 MIPS each) to obtain a representative time distribution. These results (in microseconds) and the assembly language instruction count for each operation are shown in Table 1. The Typical times are both the median and mode of the resulting distributions. All of the operations in the table include an overflow check, except that the overflow check for Interrupt Entry is postponed until the corresponding Interrupt Exit.

The results of these experiments show that low timing overheads can be achieved in practice. The typical operation times of 10 to 15 microseconds from Table 1 are cheap in comparison to other costs in the system. For example, the fastest context switch path (user mode to user mode) takes 320 microseconds on this machine, but the timing code accounts for about 36 microseconds, or 11.25%. This is the worst case; for a reasonable system operation time of about 1 millisecond, the corresponding overhead is about 3.5%, and the percentage decreases when user-mode computation time is considered (e.g. for an average user computation time of 3 milliseconds between system operations, the overhead is less than 1%). These small overheads are more than justified by the orders of magnitude increase in



Operation	Instructions	Min	Max	Average	Typical
Trap Entry	15	13.5	27.5	14.1	13.5
Trap Exit	15	12.5	26.5	13.2	12.5
Interrupt Entry	12	10.5	21.5	10.7	10.5
Interrupt Exit	15	13.5	28.5	14.0	13.5
Context Switch	13	9.5	24.5	11.9	10.5

Table 1: Multimax Timer Switch Costs (Instructions and Microseconds)

accuracy and absence of distortion in the results produced by Mach's timing facility<sup>3</sup>.

These overhead measurements generalize to other architectures, but care must be exercised in making the generalizations. The time per instruction is not uniform because it takes 2.5 microseconds to obtain a timestamp. For example, the 10.5 microsecond Typical time for the Switch operation in the Context Switch path consists of 8 microseconds to execute 12 instructions plus 2.5 microseconds to obtain the required timestamp. The Trap Entry and Trap Exit code sequences assume that the trap is known to be from user mode; two extra instructions are needed to determine this when it is not known. Similarly the Interrupt Entry and Exit sequences assume that all interrupts are blocked; two extra instructions would be needed if this is not the case. The Context Switch code also uses an extra instruction that references memory due to a compiler convention that imposes a limit of 3 scratch registers.

## 6 Conclusion

This paper has shown that the goals of low overhead and accuracy for a processor usage timing facility are not mutually exclusive in the presence of a minimal level of hardware support. By comparison to statistical timing techniques, the Mach timing facility achieves orders of magnitude improvements in accuracy and all but eliminates distortion with an overhead cost measured in single digit percentages (or less). Only minimal hardware support is required; a single free running counter for the entire machine is

<sup>3</sup>See Appendix A of [5] for details on the potential inaccuracies and distortion of statistical timing.

sufficient. An alternative that avoids contention is to provide each processor with its own counter; the facility does not require these counters to be synchronized. Hardware designers should provide this level of support, as the resulting accurate timing functionality is of great benefit to both the operating system and users. A single counter in a page of physical memory that can be mapped into applications (by software) is desirable because it makes low overhead timestamps available to applications as well as the operating system.

## A Appendix: Timer Interface

This appendix describes the interface to the module that implements the accurate timers within the kernel. It is only available within the kernel; applications can obtain usage times from the timers via the *task\_info* and *thread\_info* kernel operations.

Each thread has both a user-mode and a system-mode timer, and there is at least one set of kernel overhead timers (one per processor). In addition to the timer data structures, this module also maintains a data structure that indicates the currently running timer for each processor in the system (there is always exactly one running timer for each processor at any time).

### A.1 Initialization Routines

The first processor to boot calls `init_timers()` to initialize the kernel overhead timers and start the kernel overhead timer for the first processor. Subsequent processors call `start_timer(timer)` on their overhead timer to start it. The current implementation does not include a termination routine to stop the running timer on a processor that exits the system.

### A.2 Trap and Context Switch Routines

These routines are called by the trap and context switch code to switch the running timer. They are all versions of the **Switch** operation; stop one timer and start another using the same timestamp. The code that calls these routines is responsible for locking out interrupts and taking a timestamp. In addition, the code that calls the trap routines must determine that the trap (or return) is from (or returning to) user mode, because no timer change is needed by system traps (or returns from them). This allows implicit architectural-specific knowledge to be used (e.g., floating point error

traps must be from user mode because the kernel does not use floating point). These routines are good candidates for inline assembly language implementation.

There are two trap timing routines. In these descriptions, `ts` is a timestamp obtained by the caller after interrupts were disabled. Interrupts must remain completely disabled during the execution of these routines.

`time_trap_uentry(ts)` – Trap entry timing for traps from user mode. Checks current (user) timer for overflow.

`time_trap_uexit(ts)` – Trap exit timing for returns to user mode. Checks current (system) timer for overflow.

The interrupt timing routines require more to be done by their callers. The interrupt entry routine requires its callers to determine the new timer to switch to. This will often be the kernel overhead timer for the current processor, but additional timers can be used to measure the time spent on different interrupts. It is an error for a timer to be running on more than one processor simultaneously. The kernel overhead timers are implemented as a per-processor array to comply with this restriction. The interrupt entry routine returns the timer that was running when it was called; this must be stored (e.g., pushed onto the stack), and provided as an argument of the corresponding call to the interrupt exit timing routine.

There are two interrupt timing routines. As before, `ts` is a timestamp obtained by the caller after interrupts were disabled. Interrupts must remain completely disabled during the execution of these routines.

`time_int_entry(ts, new_timer)` – Interrupt entry timing. `new_timer` is the new timer to switch to. The timer that was running is returned as the function value and must be saved.

`time_int_exit(ts, old_timer)` – Interrupt exit timing. `old_timer` is the timer that was running before the corresponding call to `time_int_entry` (returned as its function value). Both the interrupt timer and `old_timer` (which is switched to) are checked for overflow.

The context switch code uses `timer_switch(new_timer)` to switch to a new timer. The caller must disable all interrupts. The machine-dependent routine/macro `get_timestamp()` is used to obtain the needed timestamp. The current timer is checked for overflow.

### A.3 Basic Timer Operations

This section describes the internal routines that normalize a timer and read its value. These are the routines that incorporate the synchronization derived from Lamport's clock algorithm[4]. They are internal to the timing facility implementation and are not exported to other parts of the kernel. There are two such routines:

`timer_normalize(timer)` – Normalizes a timer value. It is called from the trap, interrupt, and context switch routines if the most significant bit in the low order accumulated time word is set in a timer.

`timer_grab(timer, save)` – The basic **Read** routine for timers. It returns a save value, which is a copy of the low and high order accumulated time fields of the timer that is guaranteed to be consistent.

### A.4 Read Routines

These routines read accumulated time values from timers. They are called by routines that will pass the results back to user applications. A `time_value_t` contains fields for seconds and microseconds, and is represented as `tv` in these descriptions:

`timer_read(timer, tv)` – Read a timer.

`thread_read_times(thread, user_tv, system_tv)` – Reads both the user and system timers for the specified thread.

These routines return normalized time values.

### A.5 Scheduler Interface

This section describes the routines used by the scheduler to obtain usage timing information. The basic routine is `timer_delta`, which is used by the optimized `TIMER_DELTA` macro. This macro is used by `thread_timer_delta`, called by the scheduler to update timing information for a thread.

`timer_delta(timer, save)` takes the difference of a saved timer value and the current one, and updates the saved value to current. The difference is returned as the function value. This routine handles the general case in which the high order save value and the high order timer value may not match.

`TIMER_DELTA(timer, save, result)` is a macro form of `timer_delta` that optimizes the common case of the high save and high timer value matching. If these values don't match, `timer_delta` is called to do this operation the slow way. The difference is returned in the `result` parameter.

`thread_timer_delta(thread)` updates the scheduler's timer information for a thread. It calls `TIMER_DELTA` on both the system and user timers for the thread, updating the corresponding save values (all in the thread's data structure). The delta values are used to keep running counts of cpu usage (unweighted) and scheduler usage (weighted by overload factor). The scheduler uses these values to compute percent cpu usage and priority for a timesharing thread, respectively.

## A.6 Timestamps and Units

This section describes the interface between the timer code and the source of timestamps. These interfaces consist of macro definitions in the machine-dependent header file `machine/timer.h`. Timestamps are obtained by the `get_timestamp()` routine or macro. The trap and interrupt timing routines expect the caller to provide their timestamp.

`TIMER_MAX` defines the value at which the timestamp source rolls over. If the difference between two timestamps is negative, this value is added to correct for the (assumed) rollover. If `TIMER_MAX` is not defined, no correction is made; this is the case for timers that roll over at their full 32-bit value (on 32-bit machines).

`TIMER_RATE` defines the rate of the timestamp source in counts per second (hertz). It is used by the scheduler to calculate percent cpu usage.

`TIMER_HIGH_UNIT` defines the number of timestamp units (`TIMER_RATE` per second) that constitute one unit for the high word of the timer. Setting this to `TIMER_RATE` causes the high word to use units of seconds.

`TIMER_ADJUST()` adjusts time units for read operations. It takes a `timer_save_t` as an argument and converts it to use low units of microseconds and high units of seconds. This is a null operation and not defined if the timer low and high units are already microseconds and seconds. The timer low unit always matches the unit of the timestamp source.

`PRI_SHIFT` and `PRI_SHIFT_2` adjust the time units for scheduler delta operations. In the case that `PRI_SHIFT_2` is not defined, the timer units are shifted right by `PRI_SHIFT` to convert to priority units. Otherwise the timer units are also shifted right by the magnitude of `PRI_SHIFT_2` and added to the previous result if `PRI_SHIFT_2`'s sign is positive, or subtracted



if negative. This approximation can come within 9% or better of any desired conversion ratio[2]. For timestamp units of microseconds, PRI\_SHIFT is 16 for the 0–127 priority range, and 18 for the 0–31 priority range. PRI\_SHIFT\_2 is not defined in either case. These values can be used to determine the appropriate shifts for other timestamp units.

## A.7 Porting Guidelines

Most of the effort in porting the timing facility involves the code to invoke the trap and interrupt timing routines. Hardware dependencies usually force this code to be written in assembly language, and care must be exercised to ensure that interrupts are disabled around calls to the timing routines, and that the trap timing routines are only called when the execution mode changes (i.e., they should not be called by system-mode traps or trap returns to system-mode). The interrupt timing routines require a timer pointer to be saved for each interrupt (and each processor); the logical place to put this is the interrupt stack frame (changing its format). Additional work is required to define the quantities described in the previous section.

After the initial implementation is debugged, additional speed can be obtained by converting the trap and interrupt timing routines to inline assembly language. This is justified by the short length of the timing routines and their invocation from critical paths. This makes the procedure call overhead significant in comparison to the overall length of the routines and an important contribution to overhead. Inlining the Multimax timing routines achieved instruction count reductions of one third or more. Compiler output for the C versions of the timer routines is a useful starting point for constructing assembly language equivalents. The kernel inline tool can be used to replace the context switch timing routine with assembly language.

## References

- [1] Roberto Bisiani and Mosur Ravishankar. PLUS: A Distributed Shared Memory System. In *Conference Proceedings: The 17th International Symposium on Computer Architecture*, pages 115–124, May 1990.
- [2] David L. Black. *Scheduling and Resource Management Techniques for Multiprocessors*. PhD thesis, Carnegie Mellon University, Department of Computer Science, Pittsburgh, PA, 1990.

- [3] Leslie Lamport. Concurrent Reading and Writing. *Communications of the ACM*, 20(11):806–811, November 1977.
- [4] Leslie Lamport. Concurrent Reading and Writing of Clocks. Technical Report 27, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, 1988.
- [5] James W. Wendorf. *Operating System/Application Concurrency in Tightly-Coupled Multiple-Processor Systems*. PhD thesis, Carnegie Mellon University, Department of Computer Science, Pittsburgh, PA, 1987. Available as Technical Report CMU-CS-88-117.



# Real-Time Mach: Towards a Predictable Real-Time System

Hideyuki Tokuda, Tatsuo Nakajima, Prithvi Rao  
School of Computer Science  
Carnegie Mellon University  
Pittsburgh, Pennsylvania 15213  
hxt@cs.cmu.edu

## Abstract

Distributed real-time systems play a very important role in our modern society. They are used in aircraft control, communication systems, military command and control systems, factory automation, and robotics. However, satisfying the rigid timing requirements of various real-time activities in distributed real-time systems often requires *ad hoc* methods to tune the system's runtime behavior.

The objective of Real-Time Mach is to develop a real-time version of the Mach kernel which provides users with a predictable and reliable distributed real-time computing environment. In this paper, we describe a real-time thread model, real-time synchronization, and the ITDS scheduler in Real-Time Mach. We also discuss the implementation issues, a real-time toolset, and the current status of the system.

## 1 Introduction

Distributed real-time systems are becoming more common as real-time technology is applied to many real-time applications[20]. However, satisfying the rigid timing requirements of various real-time activities in distributed real-time systems is getting more complex due to the distributed nature of the system.

In many cases, system designers of such complex systems lack systematic development methods and analysis tools, so they resort to *ad hoc* methods to develop, test, and verify real-time systems. For processor scheduling, for instance, the cyclic executive model which uses time line analysis to schedule real-time activities is not suitable for a distributed

environment. It is very difficult to test and tune the executive based on some changes in the task set or its timing requirements for complex real-time systems [9]. Message communication scheduling in a network is also difficult since some communication media such as Ethernet do not guarantee bounded communication delay at media access level and do not provide priority-based arbitration.

A new challenge in such real-time systems is to develop a real-time kernel which can provide users with a predictable and reliable distributed real-time computing environment. In particular, the kernel should allow a system designer to analyze the runtime behavior at the design stage and predict whether the given real-time tasks having various types of system and task interactions (e.g., memory allocation/deallocation, message communications, I/O interactions, etc) can meet their timing requirements.

CMU's ART (Advanced Real-Time Technology) group has been working on a real-time version of the Mach as well as a real-time toolset for system design and analysis. Real-Time Mach is being developed based on a version of the pure kernel [1, 7] using a network of SUN, SONY workstations, and single board target machines. Unlike the standard release 2.5 Mach, this kernel includes new real-time thread management, an integrated time-driven scheduler (ITDS), real-time synchronization, and memory resident objects. The real-time thread model is based on the ARTS real-time thread model [21, 25] and our real-time scheduling theories. Real-Time Mach was also integrated with our real-time toolset, *Scheduler 1-2-3*[23] and Advanced Real-Time Monitor, *ARM*[22].

In this paper, we describe new system facilities in Real-Time Mach and the current status. In Section 2, we first introduce the real-time thread model, real-time synchronization primitives, integrated time-driven scheduler, and support for memory resident objects. Section 3 discusses implementation issues and our solution to priority inversion problems. In Section 4, we also compare our approach and real-time thread model to other operating systems. Section 5 summarizes the development status and considers future work.

<sup>1</sup> This research was supported in part by the U.S. Naval Ocean Systems Center under contract number N66001-87-C-0155, by the Office of Naval Research under contract number N00014-84-K-0734, by the Defense Advanced Research Projects Agency, ARPA Order No. 7330 under contract number MDA72-90-C-0035, by the Federal Systems Division of IBM Corporation under University Agreement YA-278067, and by the SONY Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of NOSC, ONR, DARPA, IBM, SONY, or the U.S. Government.

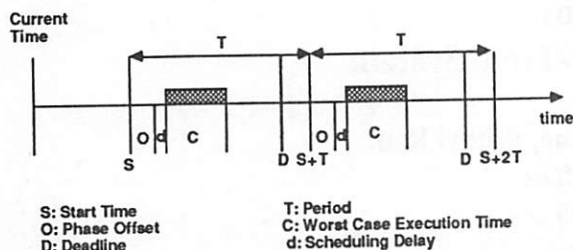


Figure 1: Timing attributes of a periodic thread

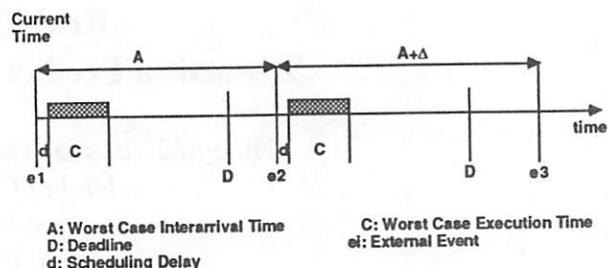


Figure 2: Timing attributes of an aperiodic thread

## 2 Real-Time Mach

The objective of Real-Time Mach (RT-Mach) is to develop a real-time version of Mach which can support a predictable real-time computing environment together with a real-time toolset. Because of the high portability of Mach, RT-Mach should be able to provide a common real-time computing environment in various machine architectures including single board computer-based targets.

In this section, we will describe the new features of RT-Mach. The current version of RT-Mach supports a real-time thread model, integrated real-time thread scheduler, policy/mechanism separation in the scheduler, real-time synchronization mechanisms, and memory resident objects.

### 2.1 RT-Thread Model

The objective of the RT-thread model is to support a predictable real-time scheduler and provide a uniform system interface to both real-time and non-real-time threads. Unlike the traditional real-time systems which often use a cyclic executive model, the RT-Mach supports an integrated time-driven scheduler [26] based on a rate monotonic scheduling paradigm [12, 13].

A thread can be defined for a real-time or non-real-time activity. Each thread is specified by at least a procedure name and a stack descriptor which specifies the size and address of the local stack region. For a real-time thread, additional *timing attributes* must be defined by a timing attribute descriptor. A real-time thread can be also defined as a *hard* real-time or *soft* real-time thread. By hard real-time thread, we mean that the thread must complete its activities by its *hard* deadline time, otherwise it will cause undesirable damage or a fatal error to the system. The soft real-time thread, on the other hand, does not have such a hard deadline, and it still makes sense for the system to complete the thread even if it passed its critical (i.e. *soft* deadline) time.

A real-time thread can be also defined as a *periodic* or *aperiodic* thread based on the nature of its activity. A periodic thread  $P_i$  is defined by the worst case execution time  $C_i$ , period  $T_i$ , start time  $S_i$ , phase offset  $O_i$ , and task's semantic importance value  $V_i$ . In a periodic thread, a new instantiation of the thread will be scheduled at  $S_i$  and then repeat the activity in every  $T_i$ . The phase offset is used to adjust a ready time within each period. If a periodic thread is a soft real-time thread, it may need to express the abort time which tells the scheduler to abort the thread. Figure 1 depicts the timing attributes of a hard periodic real-time thread.

An aperiodic thread  $AP_j$  is defined by the worst case execution time  $C_j$ , the worst case interarrival time  $A_j$ , deadline  $D_j$ , and task's semantic importance value  $V_j$ . In the case of soft real-time threads,  $A_j$  indicates the average case interarrival time and  $D_j$  represents the average response time. Abort time can be also defined for the soft real-time thread. Figure 2 depicts the timing attributes of a hard aperiodic real-time thread<sup>2</sup>.

### 2.2 RT-Thread Creation and Termination

A thread can be created, within a task, by using the *rt\_thread\_create* primitive. As we described in the model, it can be a periodic or aperiodic thread depending on its timing attributes. The timing attributes are specified in the corresponding time descriptor, and the user and kernel stack regions are also given by the stack descriptor. If a creation is successful, a unique thread id will be returned. A thread can be terminated by calling *rt\_thread\_exit* primitive. If a thread is a periodic thread, a new instantiation of the thread will be scheduled for the next start time and a new thread id will be assigned. The

<sup>2</sup>When a hard real-time thread is aperiodic, we call it a *sporadic* thread where consecutive requests of the task initiation are kept at least  $Q$  units of time apart [15].



*rt\_thread\_kill* primitive terminates the specified thread while the *rt\_thread\_wait* primitive blocks the caller thread until the target thread terminates. The *rt\_thread\_self* primitive returns the thread id of the caller. The *rt\_thread\_set\_attribute* and *rt\_thread\_get\_attribute* primitive are used to assign or get the value of the attribute respectively. The brief description of the thread attribute is shown in below.

---

```

kval_t = rt_thread_create( parent, child_thread,
                          thread_attr, entry_point, arg )
kval_t = rt_thread_exit( )
kval_t = rt_thread_kill( thread )
kval_t = rt_thread_wait( thread )
thread_t = rt_thread_self( )
kval_t = rt_thread_set_attribute( thread, thread_attr )
kval_t = rt_thread_get_attribute( thread, thread_attr )

```

---



---

```

typedef struct time_desc {
    int rt_type;                /* periodic or aperiodic thread */
    union {
        struct rt_Periodic {
            time_value_t rt_start; /* start time */
            time_value_t rt_period; /* period or response time info */
            time_value_t rt_offset; /* phase offset */
        } rt_periodic;
        struct rt_Aperiodic {
            time_value_t rt_wcia; /* worst case interarrival time */
        } rt_aperiodic;
    } rt_attribute;
    time_value_t rt_wcec; /* worst case exec time */
    time_value_t rt_deadline; /* deadline */
    time_value_t rt_abort; /* abort time */
    int rt_value; /* semantic_value */
    ...
} time_desc_t;

typedef struct stack_desc {
    vm_address_t rt_stack_addr;
    vm_size_t rt_stack_size;
    ...
} stack_desc_t;

typedef struct thread_attribute {
    time_desc_t time_desc;
    stack_desc_t stack_desc;
    ...
} thread_attr_t;

```

---

## 2.3 RT-Thread Synchronization

Synchronization among threads is necessary since all threads within a task share the task's resources. The synchronization mechanism in RT-Mach is based on mutual exclusion using a lock variable. A thread can allocate, deallocate, and initialize a lock variable. A simple pair of *rt\_mutex\_lock* and *rt\_mutex\_unlock* primitives is used to specify mutual exclusion. The *rt\_mutex\_trylock* primitive is used for acquiring the lock conditionally. A modified version of the condition variable is also created for specifying a conditional critical region. A pair of *rt\_condition\_signal* and *rt\_condition\_wait* primitives is used to synchronize over a condition variable. RT-Mach uses the earliest deadline first (or highest priority first) policy as a queueing policy in both the *rt\_mutex* and *rt\_condition* primitives. A caller can also control its priority inheritance policy by setting the proper mutex's or condition variable's attribute.

---

```

kval_t = rt_mutex_allocate( lock, lock_attr )
kval_t = rt_mutex_deallocate( lock )
kval_t = rt_mutex_lock( lock, timeout )
kval_t = rt_mutex_unlock( lock )
kval_t = rt_mutex_trylock( lock )

kval_t = rt_condition_allocate( cond, cond_attr )
kval_t = rt_condition_deallocate( cond )
kval_t = rt_condition_wait( cond, lock, cond_attr, timeout )
kval_t = rt_condition_signal( cond, cond_attr )

```

---

Unlike the ordinary mutual exclusion mechanism, the *rt\_mutex\_lock* and *rt\_mutex\_unlock* pair provide a priority inheritance mechanism in order to avoid an unbounded *priority inversion* problem. Priority inversion occurs when a high priority task must wait indefinitely for a lower priority task to execute.

Suppose that a low priority thread  $\tau_L$  is in the critical region. While thread  $\tau_L$  is executing, a high priority thread  $\tau_H$  attempts to enter the critical region by executed the *rt\_mutex\_lock* primitive. Since  $\tau_L$  is in the critical region,  $\tau_H$  must wait for  $\tau_L$  to exit. Now suppose that other threads  $\tau_{M_1} \dots \tau_{M_k}$  become active. These threads can begin their computation and will preempt thread  $\tau_L$ , thus we cannot bound the worst case blocking time of  $\tau_H$ .

In order to bound the worst case blocking time of threads, our group has developed priority inheritance protocols including *Priority Ceiling Protocol* [18]. In this example, once  $\tau_H$  executes *rt\_mutex\_lock*, then  $\tau_L$  will inherit the high priority from  $\tau_H$ . In this way, the highest priority thread's worst case blocking time is bounded by the size of critical region (See Section 2.5).

## 2.4 RT-Thread Scheduling

In real-time operating systems, thread scheduling plays an important role in managing the system resources in a timely fashion. However, traditional operating systems do not provide us a flexible and a adaptable scheduling management. We have developed a novel scheduling model, Integrated Time-Driven Scheduler (ITDS) for the ARTS kernel [24], and have extended the model for RT-Mach. This section describes the Mach scheduling mechanism and an extended ITDS model.

### 2.4.1 Mach Scheduling Mechanism

Mach provides a flexible processor allocation facility. The facility uses two objects: *processor* and *processor set*. A processor object represents a physical processor and a processor set object corresponds to a set of processors. A thread belongs to a processor set and similarly a processor belongs to a processor set. A special processor set called a *default processor set* exists. Before a new processor set is created, all processors belong to a default processor set.

The most important data structure to manage scheduling of thread is the *run queue*. All processor sets have their own respective run queues. When a thread becomes runnable, it is enqueued into the run queue of the processor set to which the thread belongs. Also, when the current running thread in a processor is blocked or preempted, the new thread is chosen from the processor set where the blocked thread belongs. Because threads do not migrate between processor sets, we can choose a suitable scheduling policy for each processor set. However, the current Mach scheduler has only *round-robin* and *fixed priority* policies and manages 32-levels of thread priorities. Mach's fixed priority policy preempts the running thread if there are runnable threads with same priority and its quantum is expired. In real-time computing, such preemption decreases schedulability and we need other scheduling policies, for example, rate monotonic policy, and various aperiodic servers [19].

### 2.4.2 ITDS Scheduling in RT-Mach

The objective of the integrated time-driven scheduler is to provide predictability, flexibility, and modifiability for managing both *hard* and *soft* real-time activities. The ITDS scheduler allows the system designer to predict whether the given task set can meet its deadlines or not.

The ITDS scheduler adopted a *capacity preservation* scheme to cope with hard and soft types of real-time activities. By bandwidth preservation we mean that we divide the necessary processor cycles between the two types. We first analyze the necessary processor cycles by accumulating the total computation time for the hard periodic and sporadic activities. Then, we will assign the remaining schedulable amount of the unused processor cycles to the soft real-time tasks.

The ITDS scheduler was designed and implemented using an object model and layered structure. The scheduling policy

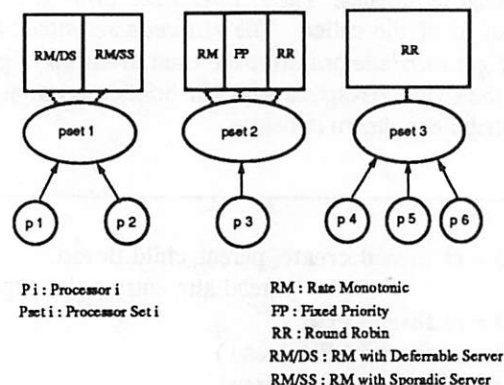


Figure 3: ITDS for RT-Mach

used by the scheduler object is a self-contained object and separated from the mechanism layer used to actually switch threads within the kernel. Using the ITDS scheduler, we can assign five different policies for each processor set in RT-Mach (See Figure 3).

The low level mechanism layer is divided into two sublayers: a processor set management sublayer and a thread dispatching management layer. The processor set management sublayer manages context switching, preemptions of threads and assignment of processors. The thread dispatching management layer controls idle threads and aperiodic servers: the polling server, deferrable server, and sporadic server [19].

Each processor set can have a scheduling policy since each processor set maintains its own run queue and operations for controlling the run queue. Therefore, we can configure the system for the various real-time applications. For example, let us consider the multiprocessor with six processors, and three processor sets: *pset*<sub>1</sub>, *pset*<sub>2</sub>, and *pset*<sub>3</sub>. *pset*<sub>1</sub> has two processors: *p*<sub>1</sub> and *p*<sub>2</sub>, *pset*<sub>2</sub> has one processor: *p*<sub>3</sub>, and *pset*<sub>3</sub> has a three processors: *p*<sub>4</sub>, *p*<sub>5</sub> and *p*<sub>6</sub>. *pset*<sub>1</sub> executes real-time applications using two processors using rate monotonic with deferrable server or rate monotonic with sporadic server. *pset*<sub>2</sub> executes real-time and non real-time applications, so the application program may change scheduling policies from rate monotonic, fixed priority, or round-robin. *pset*<sub>3</sub> executes non real-time applications using three processors by round-robin algorithm.

We can change scheduling policies by using the following primitives. The policy attribute is used to pass policy specific arguments such as a server's period, capacity of the server, etc. to the specified policy module in the system.

---

```
kval.t = rt_get_sched_policy( policy, policy_attr )
kval.t = rt_set_sched_policy( policy, policy_attr )
```

---

## 2.5 Memory Object Management

The kernel must also avoid unbounded delay while it manages memory objects for real-time threads. In general, Mach's resource allocation policy is based on *lazy* evaluation technique. For instance, if a thread allocates a region of memory, the system does not allocate the physical memory object unless the thread touches the region and cause a page fault.

In order to eliminate such unpredictable page fault handing delay, a real-time thread can "pin-down" any region of its parent task's virtual address space by using the following *vm\_wire* primitive.

---

```
kval.t = vm_wire(task, start_addr, size, access_type)
```

---

## 2.6 Schedulability Analysis

In the RT-Thread Model, our goal is to provide a better interface to adopt the well-known schedulability analysis techniques. For instance, given a set of periodic, independent tasks in a single processor environment, with the rate monotonic scheduling algorithm the worst case schedulable bound is 69% [13], the average case is 88% [12], and the best case, where threads have harmonic periods, is up to 100% of the CPU utilization.

In the case of a more general task set where threads can synchronize via critical regions, we can also bound the synchronization (blocking) time for each task by using the priority ceiling protocol. Using these inheritance protocols, we can also check schedulable bound for  $n$  periodic threads as follows.

$$\forall i, 1 \leq i \leq n, \frac{B_i}{T_i} + \sum_{j=1}^i \frac{C_j}{T_j} \leq i(2^{\frac{1}{i}} - 1)$$

where  $C_i$ ,  $T_i$ ,  $B_i$  represents the total computation time, the period, and the worst case blocking time of *Thread<sub>i</sub>* respectively.

These techniques are integrated with our real-time toolset. However, providing end-to-end schedulability analysis for a

given task set which communicates over a real-time network still remains as a future challenge<sup>3</sup>.

## 3 Implementation

The current version of RT-Mach is being developed using a network of SUN, SONY workstations and single board target machines. We first experimented with our real-time thread model using a modified version of Release 2.5 Mach kernel which can support fixed priority thread scheduling, a cpu server (i.e., processor set), and a *vm\_wire* call. We then moved to the current pure kernel based environment. The pure kernel provided us much better execution environment where we can reduce unexpected delays in the kernel and can run real-time threads without having a UNIX server, if necessary. The pre-emptability of the kernel was also improved significantly since many device drivers are no longer in the kernel.

A platform for RT-Mach has slightly different requirements in terms of its execution environment. For embedded real-time applications, we need to support not only various types of workstations, but also a wide variety of single-board or multiple-board based target machine environments. Booting the target machines also requires a different booting procedure via the system's backplane-bus or via a network. For instance, we are working on a VME-bus based target environment. For real-time communication, we are also supporting a FDDI network in addition to the IEEE 802.5 token ring network.

In this section, we will describe some implementation issues encountered in our first version of the pure kernel-based RT-Mach. We also discuss the capability of the real-time toolset and the current status of RT-Mach.

### 3.1 Implementation Issues

Our primary focus in the current implementation was to remove unbounded delay in the system and provide better pre-emptability among real-time threads. However, there are many places we encountered problems in managing system resources in Mach. In many cases, we can recognize differences in resource allocation policy between the time-sharing paradigm and the real-time computing paradigm. Major policy differences are

- "lazy" evaluation vs. "eager" evaluation policy
- FIFO ordering vs. deadline-driven ordering (starvation-free vs. no missed deadlines)
- unbounded delay vs. bounded delay

---

<sup>3</sup>Using the latest processor architecture such as various RISC chips, it becomes an interesting practical problem in determining the worst case execution time for sections of code (which must take into account cache management and pipelined architectures).



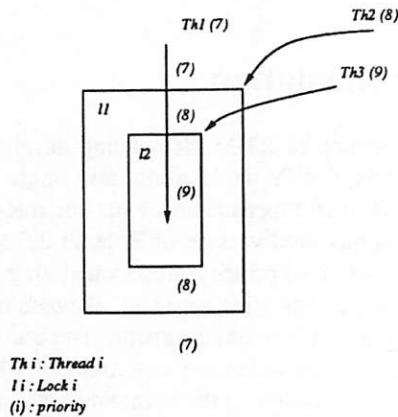


Figure 4: Priority Inheritance in Nested Lock

In many resource allocation cases, Mach takes advantage in deferring the actual allocation of resources until the requester needs it. *Copy-on-write* and *map-on-reference* techniques are good examples of the lazy evaluation scheme. On the other hand, this type of memory management policy often creates an unpredictable delay in getting the actual resources that the requester needs. For instance, if a thread needs to allocate memory, it calls *vm\_alloc* routine, but the actual physical memory may not be allocated unless the thread actually touches that region and causes a page fault. However, if the thread is a real-time thread, it cannot afford to wait for unpredictable page fault service time. Rather, we would like to allocate memory resources in eager fashion. We can then estimate the worst case time for the allocation delay.

Mach uses many queues to manage various system resources such as ready queues, message queues, and free memory list queues. FIFO queuing is often used in these queues since the system can easily avoid the starvation among waiting threads. However, in a real-time environment, FIFO queuing often creates a priority inversion problem. If all of real-time threads can meet their deadlines, then there will be no starvation among these threads.

Similarly, in real-time synchronization, we do not treat waiting threads in FIFO order. For instance, when a real-time thread attempts to enter a critical region, it will be queued in its waiting queue in earliest deadline first (or the highest priority first) order. Then, when a thread exits from the critical region, the highest priority real-time thread will be chosen rather than the oldest waiting thread in the waiting queue.

In RT-Mach, we also use a basic priority inheritance protocol to avoid priority inversion problem in real-time synchronization. Let us describe three interesting cases where priority inheritance requires an additional mechanism.

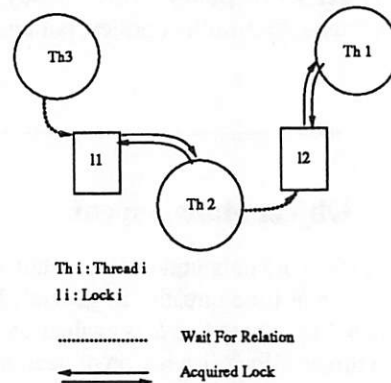


Figure 5: Cascaded Priority Inheritance

In the first example, when a priority inversion occurs in a nested critical region, the propagation of the effective priority has to be performed carefully. Let us consider three threads:  $th_1$ ,  $th_2$ , and  $th_3$ , and two locks:  $l_1$  and  $l_2$ . We assume that  $th_1$  has priority 7,  $th_2$  has priority 8, and  $th_3$  has priority 9. The larger value means higher priority.  $th_1$  acquires  $l_1$  and then  $l_2$ .  $th_3$  then tries to acquire  $l_2$ , and  $th_2$  tries to acquire  $l_1$ . The priority inheritance protocol makes  $th_1$  inherit a priority of  $th_3$ , then the priority of  $th_1$  becomes 9. After  $th_1$  unlocks  $l_2$ , the priority of  $th_1$  must become 8 because  $th_2$ 's priority is 8 and  $th_2$  waits for  $th_1$  for unlocking  $l_1$ . When  $th_1$  unlocks  $l_1$ , the priority of  $th_1$  goes back to the base priority 7.

The second case is where the *waiting for* relation must be maintained among threads so that we can propagate the proper priority to a target thread. Let us consider three threads:  $th_1$ ,  $th_2$ , and  $th_3$ , and two locks  $l_1$  and  $l_2$ .  $th_3$  has the highest priority,  $th_2$  has middle priority, and the priority of  $th_1$  is the lowest.  $th_1$  acquires  $l_2$ .  $th_2$  acquires  $l_1$  and waits for unlocking  $l_2$ . When  $th_3$  tries to acquire  $l_1$ ,  $th_1$  must inherit the priority of  $th_3$ . To manage this type of blocking case, the effect of priority inheritance is cascaded<sup>4</sup>.

The third case is where the highest priority thread is timed out while waiting for a lock. For instance, in Figure 5,  $th_3$  is inherits the priority of  $th_1$ . However, if  $th_1$  is timed out while waiting for  $l_1$ , the priority of  $th_3$  must be changed back to the priority of  $th_2$ .

The above problems can be solved by maintaining a relation between the lock variables and the threads. In RT-Mach, each thread maintains a pointer to lock variables, and the lock variable also keeps track of the nesting relation to the other locks and the holding thread. For example, in Figure 4, when

<sup>4</sup>The similar cascading problem and its solution was also described in [8]

$th_1$  acquires  $l_1$ ,  $th_1$  points to  $l_1$ . Next, when  $th_1$  acquires  $l_2$ ,  $l_2$  points to  $l_1$ , and  $th_1$  points to  $l_2$ . Then, if  $th_1$  unlocks  $l_2$ , the priority of  $th_1$  can degrade to the priority of  $th_2$  because  $l_2$  knows  $l_1$ , and  $l_1$  knows the priority of  $th_2$ .

We can solve the timeout problem mentioned above, the following way. From Figure 5,  $th_1$  points to  $l_1$ ,  $l_1$  points to  $th_2$ ,  $th_2$  points to  $l_2$ , and  $l_2$  points to  $th_3$ . So,  $th_1$  can inherit the priority of  $th_3$ . If  $th_1$  is timed out while waiting to acquire  $l_1$ , the priority of  $th_2$  reverts to its base priority, and  $th_3$  inherit that priority.

### 3.2 Real-Time Toolset

We have also developed a set of tools which we can use in conjunction with Real-Time Mach for predicting the behavior of the system and for runtime monitoring and debugging. The goal of the toolset is to incorporate a system-wide scheduling analysis which includes communication and synchronization among real-time threads. The toolset consists of *Scheduler 1-2-3* and *ARM*.

*Scheduler 1-2-3* is a schedulability analyzer and is an X11-window based interactive tool for creating, manipulating, and analyzing real-time task sets. It employs methods ranging from closed form analysis to simulation to determine whether a feasible schedule exists for a given task set and what the schedulable bound is for that set.

*ARM* (Advanced Real-Time Monitor) is also an X11-window based tool designed to analyze and visualize the runtime behavior of the target nodes in real time. The ARM allows us to reach into a remote target and view the scheduling events which are extracted using event taps in RT-Mach.

### 3.3 Current Status

In the current version of RT-Mach, the RT-thread model and extended ITDS scheduler have been implemented. RT-Mach has also been integrated with the real-time tool set. In Figure 6, we show the snapshot of ARM with three periodic threads, and ten aperiodic threads. ARM is useful for monitoring the occurrences of preemption and the order in which threads are executed. Timing bugs can also be deleted easily using ARM.

Figure 7 demonstrates the RT-thread model. In the gmol demo, seven periodic threads are created and every thread represents an atom of a molecule. The threads, while executing, cause the molecule to rotate about an axis passing through the atom at the center. If scheduled correctly, the molecule maintains its integrity, otherwise the atoms move in a random fashion. Gmol visually demonstrates lack of schedulability, when the molecule's rotation is random. If we use a proper scheduling policy, we can ensure schedulability with high CPU utilization and each molecule rotates in a completely synchronized way (the left figure). However, if the scheduling policy is inappropriate, some deadlines are missed, so the synchronized rotation of the molecules is violated, and each molecule rotates

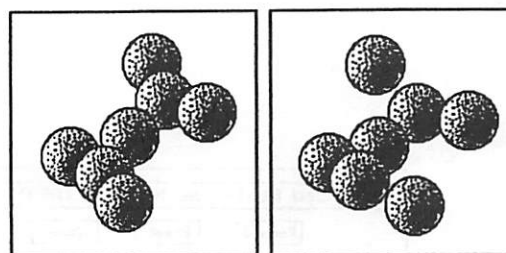


Figure 7: The Snapshot of Gmol

Null Argument Trap	0.03 ms
Null Argument MIG	0.30 ms
Context Switch	0.26 ms
vm_alloc 1KB (no wiring)	0.59 ms
vm_alloc 1KB (wiring)	2.67 ms
Port Allocate/Deallocate	1.2 ms
RT-Thread Creation/Termination	4.258 ms
RT-Lock and Unlock	0.146 ms

Table 1: The Basic Performance of RT-Mach

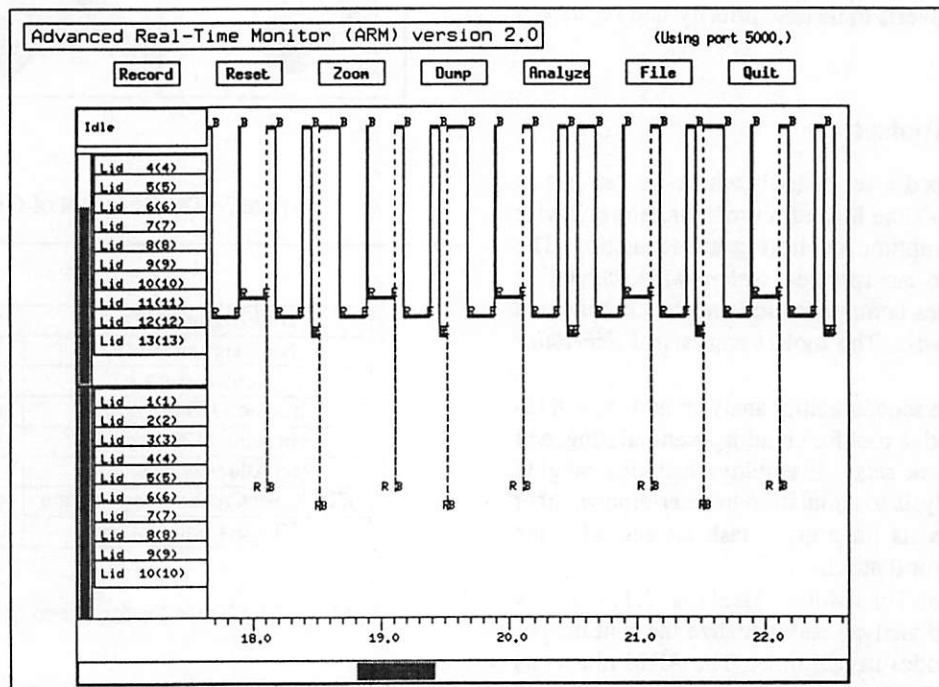
in chaotic manner (the right figure). Gmol further demonstrates the importance of the RT-thread model in preserving the schedulability of RT-thread. The traditional system offers the *delay* primitive for representing time description. However, the calculation of delay time and the execution of delay primitives may not be atomic giving rise to invalid time values, resulting in non synchronized rotation of the molecule.

Table 1 summarizes the basic performance of the current version of RT-Mach.

All measurements are performed on a Sun3/60 workstation with 12 Mbytes by repeating the target function more than 10,000 times. The trap interface to the kernel is about 10 times faster than MIG<sup>5</sup>. The context switch time between threads is acceptable in our application, and we can assume this number as the worst case since all resources are wiring down for these threads. Allocation of memory is normally done without wiring, thus the overhead is relatively small. However, additional wiring cost is not reduce to negligible yet, since we simply reused the original wiring facility in Mach. The creation and termination cost are mainly due to allocation and deallocation of system resources which belongs to a thread. The total cost should be reduced further by re-

<sup>5</sup>Mach uses a MIG (Mach Interface Generator) [10] to call kernel primitives in a object-oriented fashion and generates stubs for user programs.





```

Time: 18850 ms.
Periodic tasks : 1
Aperiodic tasks : 10
Total CPU utilization: 0.533 (cyclic tasks (0.444)) (acyclic tasks (0.089))
Meet deadline : 64
Missed deadline: 0
  aborted: 0
Events: 1088 events 57.719 per second

```

This figure shows an example of the history execution diagram. The top six boxes indicate the action menus. The top half of threads correspond to the periodic threads and the bottom half correspond to the aperiodic threads. Character 'R' shown in the execution history diagram indicates a periodic thread which becomes runnable and 'E' indicates that it terminates with its deadline being met; 'A' or 'C' indicate that the thread is aborted or canceled due to a missed deadline. 'B' indicates that the thread is blocked waiting for some event, or for preemption. The bottom window shows the various statistical information.

Figure 6: An Example of the ARM Snapshot

placing the allocation and deallocation policies. The current version of the `rt_thread_create`, `rt_thread_exit`, `rt_mutex_lock`, and `rt_mutex_unlock` primitives are implemented using a trap mechanism, rather than MIG.

## 4 Related Work

The pure kernel-based approach is gaining popularity and several pure (or micro) kernel-based operating systems have been developed for the distributed computing environment [3, 16, 17]. Advantages of using a pure kernel instead of a standard monolithic kernel is that the preemptability of the kernel will be inherently better, the size of the kernel becomes much smaller, and modification of the kernel will be easier. However, only a few micro-kernels were designed for supporting distributed real-time applications.

In many commercially available real-time operating systems and executives, a fixed priority-based preemptive scheduling policy has been used. Emphasis was placed on fast interrupt latency, fast context switching, and small kernel size [6]. Although these factors are important properties for real-time operating systems, users were often forced to create an *ad hoc* scheduling module for each particular application. Further more, under a transient overload, users may lose control over which tasks should complete their computations and which should be aborted or canceled. It is also difficult to remove priority inversion problems in the kernel and bound the worst case blocking time for threads.

The proposed real-time thread model is different from many other thread models. In particular, our model

- distinguishes between real-time and non real-time threads,
- assumes explicit timing constraints for each real-time thread, and
- provides a priority inheritance protocol to avoid unbounded priority inversion.

The POSIX-Thread proposal[11] is very similar to Mach's C-Thread package[5] and it also does not distinguish between real-time threads and non-real-time threads. This poses a problem of identifying the type of threads that can "pinned down" its memory objects. However, it can dynamically select the thread scheduling policy and a thread also contains the thread attributes such as "inherit priority", "scheduling priority", "scheduling policy", and "minimum stack size". Thus, adding the timing attributes would be very simple.

The Ultrix-Thread model[4] does not address real-time thread issues, however, the designer intended to create much lighter threads by leaving the context information of thread at the process level as much as possible. Thus, creation of a new thread can be done by specifying thread's stack page and guard page address: `tfork( stack_ptr, guard_ptr )`.

The Topaz-Thread model[14] provides a clean thread interface library at the Modula-2+ language level, however, it does not address real-time thread issues.

## 5 Summary

The objective of Real-Time Mach is to develop a real-time version of Mach which can support a predictable real-time computing environment together with a real-time toolset. In particular, the kernel should allow a system designer to predict the schedulability of *hard* and *soft* real-time tasks which communicate over a real-time network.

In this paper, we described a real-time thread model, real-time synchronization, integrated time-driven scheduler, and memory resident objects for Real-Time Mach. We also discussed the implementation issues, real-time toolset, and the current status of the system.

We are still improving the system capability in order to provide a system-wide schedulability analysis in Real-Time Mach. In particular, we are working on predictable real-time communication support, priority inversion problems in Mach IPC, and multi-board based multiprocessor targets.

## 6 Acknowledgments

We would like to thank the members of the ART Project and the Mach group for their valuable comments and inputs to the development of Real-Time Mach.

## References

- [1] M.J. Accetta, W. Baron, R.V. Bolosky, D.B. Golub, R.F. Rashid, A. Tevanian, and M.W. Young, "Mach: A new kernel foundation for unix development", *In Proceedings of the Summer Usenix Conference*, July, 1986.
- [2] David L. Black, "Scheduling support for concurrency and parallelism in the Mach operating system", *IEEE Computer*, Vol.23, No.5, 1990
- [3] D.R. Cheriton, G.R. Whitehead and E.D. Sznyter, "Binary emulation of UNIX using V Kernel", *In proceedings of Summer Usenix Conference*, June, 1990.
- [4] D. S Conde, F. S. Hsu, and U. Sinkewicz, "Ultrix threads", *In Proceedings of Summer Usenix Conference*, June, 1989.
- [5] E. C. Cooper, and R. P. Draves, "C threads", Technical report, Computer Science Department, Carnegie Mellon University, CMU-CS-88-154, March, 1987.
- [6] B. Furht, J. Parker, and D. Grostick, "Performance of REAL/IX<sup>TM</sup> - Fully Preemptive Real Time UNIX", *Operating System Review*, Vol.23, No.4, April, 1989

- [7] D. Golub, R. Dean, A. Forin, and R. Rashid, "Unix as an application program", *In the proceedings of Summer Usenix Conference*, June, 1990.
- [8] Mark Heuser, "An implementation of real-time thread synchronization", *In Proceedings of Usenix Summer Conference*, June, 1990.
- [9] P. Hood and V. Grover, "Designing real time systems in ADA", Tech Report 1123-1, SofTech, Inc., January, 1986.
- [10] M.B. Jones, and R.F. Rashid, "Mach and Matchmaker: Kernel and language support for object-oriented distributed system", *In proceedings of the first conference of OOPSLA*, September, 1986
- [11] IEEE, "Realtime Extension for Portable Operating Systems", P1003.4/Draft6, February, 1989.
- [12] J. P. Lehoczky, L. Sha, and Y. Ding, "The rate-monotonic scheduling algorithm: Exact characterization and average case behavior", Department of Statistic, Carnegie Mellon University, 1987.
- [13] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment", *Journal of the ACM*, Vol.20, No.1, 1973.
- [14] P. McJones and Swart P, "Evolving the unix system interface to support multithreaded programs", Technical report, Tech Report 21, Part I, DEC SRC, September, 1987.
- [15] A. K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment", *PhD thesis*, Massachusetts Institute of Technology, May 1983.
- [16] S.J. Mullender, G.V. Rossum, A.S. Tanenbaum, R. Renesse and H. Staveren, "Amoeba: A Distributed Operating System for the 1990s", *IEEE Computer* Vol.23, No.5, May, 1990
- [17] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser, "Chorus distributed operating system", *Computing Systems Journal*, The Usenix Association, December, 1988
- [18] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization", Technical Report CMU-CS-87-181, Carnegie Mellon University, November 1987
- [19] B. Sprunt, L. sha and J. P. Lehoczky, "Aperiodic Task Scheduling for Hard-Real-Time Systems", *The Journal of Real-Time Systems*, Vol.1, No.1, 1989.
- [20] J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems", *IEEE Computer*, Vol.21, No.10, October, 1988.
- [21] H. Tokuda and M. Kotera, "A real-time tool set for the ARTS kernel", *Proceedings of 9th IEEE Real-Time Systems Symposium*, December, 1988.
- [22] H. Tokuda and M. Kotera, "Scheduler1-2-3: An interactive schedulability analyzer for real-time systems", *In Proceedings of Compsac88*, October 1988.
- [23] H. Tokuda, M. Kotera, and C. W. Mercer, "A real-time monitor for a distributed real-time operating system", *In Proceedings of ACM SIGOPS and SIGPLAN workshop on parallel and distributed debugging*, May, 1988.
- [24] H. Tokuda, M. Kotera, and C. W. Mercer, "An integrated time-driven scheduler for the ARTS kernel", *In Proceedings of 8th IEEE Phoenix Conference on Computers and Communications*, March, 1989.
- [25] H. Tokuda and C. W. Mercer, "ARTS: A distributed real-time kernel", *ACM Operating Systems Review*, Vol.23, No.3, July, 1989.
- [26] H. Tokuda, C. W. Mercer, Y. Ishikawa, and T. E. Marchok, "Priority inversions in real-time communication", *In Proceedings of 10th IEEE Real-Time Systems Symposium*, December, 1989.

# Developing Benchmarks to Measure the Performance of the Mach Operating System \*

David Finkel      Robert E. Kinicki      Aju John  
Bradford Nichols      Somesh Rao

Department of Computer Science  
Worcester Polytechnic Institute  
Worcester, MA 01609

## Abstract

We have developed a series of benchmark programs to test the performance of the Mach operating system. A discussion of the philosophy of the benchmarks is given, and the individual benchmark programs are described. Results of the benchmarks on 386-based machines running Mach 2.5, Mach 3.0, and SCO System V/386 are given.

## 1 Introduction

As Mach progresses towards its micro-kernel version, important questions about its performance need to be answered. The micro-kernel version will have some clear advantages over the monolithic kernel version: removal of contention points, improved portability, scalability, and ease of maintenance. However, an important question is how much these changes will affect performance. The micro-kernel version of Mach must demonstrate good performance if Mach is to gain acceptance outside the research community.

In order to understand the performance implications of this micro-kernel design, we have undertaken a performance study of Mach. The overall goal

---

\*This research is supported by a grant from the Research Institute of the Open Software Foundation.

of the study is to develop a series of benchmarking programs which simulate system usage at a user level, and will allow system developers to evaluate the performance of the evolving versions of Mach.

We are in the process of developing a suite of benchmark programs to test the performance of Mach on a variety of platforms. In the following sections, we describe our approach to developing system benchmark programs, describe the benchmark programs we have developed so far, and give some results for these benchmarks run on identical 386-based systems running Mach 2.5, Mach 3.0 and SCO System V/386. In the last section we present some conclusions and outline our plan for future work.

## 2 Previous Work

Of course, a large number of benchmarking studies have been published. Mach-specific benchmark results have appeared in [BLA89], [FOR89], [GOL90], and [TEV87]. For the most part, these performance studies were low-level benchmarks, repeatedly exercising a single system call or system service. While such low-level benchmarks are important to system developers to test the efficiency of their implementations, the results may be difficult for users to interpret in terms of the kinds of applications they are interested in running.

We identified a large number of benchmarking programs and suites, for example [SPE90], [CUR76], [DON87], [SMI90], and [WEI84]. For the most part, these benchmarks emphasize CPU intensive applications, and did not specifically target operating system performance. An especially thorough set of benchmarks for Unix systems is given in [ACE89]. This report describes ten benchmark suites, and gives the results of running them on 47 Unix systems. Again, most of the benchmarks are either CPU-bound, or low-level tests of system functions.

## 3 Developing User-Level Benchmarks

Our goal is to create user-level benchmarks of system services. We wish to create benchmarks that make extensive use of significant system services in a mix reflecting the usage by actual user applications. We could have accomplished this either by adopting actual user code, as in the SPEC benchmarks



[SPE90], or by writing synthetic programs with the desired properties. We have adopted the latter strategy, for the following reasons.

First, writing synthetic programs allowed us to understand in detail what system services are used by the benchmark programs. This in turn enables us to create low-level benchmarks to test individual system functions to understand the reasons for the differences between results on different systems, and to provide guidance to system developers about areas of the system needing improvement. Second, synthetic programs permitted us to parametrize the benchmarks, to allow the same benchmark program to run on small scale and large scale systems (with different parameter settings). Third, writing synthetic programs allows us to make sure that our suite of benchmarks collectively covers the entire range of important systems services. Finally, we can ensure that our synthetic benchmarks are easily portable and available free of cumbersome licensing requirements.

We used several methods to ensure that our synthetic benchmark programs reflected the usage pattern of system services representative of actual user application programs. One approach was to run actual user applications under the control of a profiler, such as *gprof* [GRA82]. This allowed us to identify the particular system calls used by the program, and the number of times each call was made in the application. We also used statistical utilities, such as *vmstat* and *iostat*, to track the use of other system resources. This then gave us some guidelines in constructing our programs and in tuning them to match the resource utilizations of the original programs.

Another approach to constructing our benchmarks was to examine the source code of the user application, and identify the key system calls. This, together with the information provided by the statistical utilities, allowed us to construct our benchmarks. In addition, one of our benchmarks does not emulate a user program, but is a specially constructed test program designed to test specific operating system features.

At present our benchmark programs all issue only Unix system calls. By avoiding Mach-specific system calls, the benchmarks are directly portable to Unix systems and the timing comparisons reflect differences in implementations and not differences in system calls.

## 4 Descriptions of Benchmarks Programs

At this writing we have completed three benchmark programs. They are 1) the synthetic dump benchmark, which is intended to emulate the activities of the Unix *dump* program, 2) the synthetic compilation benchmark, which emulates the activities of the *gcc* compiler compiling a large program, and 3) the jigsaw puzzle program, which tests memory management efficiency. We describe each of them in detail below. In addition, in Section 6 we discuss additional benchmarks which are under development.

### 4.1 Synthetic Dump Benchmark (*sdump*)

*sdump* is a synthetic benchmark program that functionally simulates the working of the Unix *dump* program, which is used to backup a file system onto tape. Executing *dump* requires root privileges and it is not possible to backup a part of the file system or a sub-directory. The *dump* program creates child processes to write the data on the tape unit and these processes communicate to the parent through pipes to obtain data and control information. *dump* speeds up the process of file transfer by forking up to a maximum of twenty processes that independently transfer data from disk to tape.

The key issue considered during the design of *sdump* was to avoid dumping a file-system. Instead, *sdump* uses a directory containing a fixed number of files in place of a file-system. It was done this way for the following reasons: 1) It is easier to transport a directory to systems, than an entire file-system. 2) This enables any user to run this benchmark, as opposed to the *dump* program, which requires super-user privileges. 3) A fixed number of files of known sizes can be used to test and compare a set of machines. 4) Varying the number of files permits the benchmark to be scaled for testing a variety of machine types.

A file that can be locked was used to simulate the tape backup unit to avoid making the test results depend on the speed of a particular tape unit. File locks were used so that at any instant, only one process could write to it, thus simulating the sequential nature of a tape device. Other processes will be blocked while the simulated tape is in use.

As in the *dump* program, *sdump* forks a child process that writes data onto the simulated tape. The parent sends the data through a pipe, to the child in blocks of 4 kilobytes (KB), which is the buffer size of the pipe. The

child reads the pipe to gather data and writes it in 4KB blocks onto the file emulating the tape. The child then checks a control pipe to see if more data is to come from the parent. A rudimentary protocol was established within *sdump* to synchronize the data flow. When the parent has finished sending the last packet of data, it sends a special character to indicate the end of transmission. This is recognized by the child which closes the file emulating the tape and exits.

Some aspects of *dump* are not simulated by the *sdump* benchmark. *sdump* has no operation in it to simulate the process of traversing the inodes in a file system, checking for dates and bit-mapping the pattern of inodes that need to be dumped, onto an array. *dump* depends on this array to determine which inode blocks need to be dumped. *sdump* also does not synthesize the *dump*'s feature, since tape unit failures will not occur in *sdump*.

*sdump* has the optional ability to fork a separate process for each file in the directory, each of these can then independently read the data from its file from the disk. Each of these processes can in turn create its own child process to dump the data on the tape. Multiple processes are created and each of them competes for the file that emulates the tape. The process that gains control first locks the unit for its use, while others wait for the resource. In addition, the size of the data packets written to pipes can be increased and the rate of reading the pipes can be reduced to stress the system to higher limits. The results shown are for the case when *sdump* does not fork individual processes for each file.

Depending on the operating system under test, time is measured by using the *ftime* or *gettimeofday*.

## 4.2 Synthetic Compilation Benchmark (*scomp*)

The purpose of this benchmark is to utilize system resources in a manner similar to a compiler. The approach taken in creating the benchmark was to use monitors and profilers to obtain information about system resources utilized as the compiler operated on a well-defined set of input files. The data collected was used to write a program called *scomp* which mimics the actual compiler in measured usage of system resources.

The compiler used was gnu's *gcc*. System resource utilization was measured for *gcc* compiling itself. This combination was chosen because *gcc* compiling *gcc* is a commonly used benchmark and because we had access to

the source code, which was necessary for using the profiler.

Profiling was done using the *gprof* program on a Sun 3/60 workstation. Using *gprof* required that *gcc* be built first with a *-g* option during compilation to create a profiled version and then making *gcc* again using the profiled version of itself as the compiler. The profiled version of *gcc* creates data files used by the *gprof* program to tally procedure call tables and a call graph. From the procedure call tables the system calls were identified.

Additional information about the amount of memory utilized and the amount of paging activity was collected using the *vmstat* monitor. The average number bytes for read and write calls was not readily available so benchmark values were chosen based on the number of files accessed, the number of calls and the average size of source and intermediate files (.i, .s, .o) used. The relative portion of time spent inside system space to inside user space was measured using the time program.

When the *gcc* compiles *gcc*, a make program executes *gcc* in sequence to compile each source file and then to link them. Each time *gcc* is called it forks a child process and executes in sequence the precompiler, compiler, assembler and the linker, depending on the input flags or the type of files supplied.

In order to maintain the environment of process forking and program execution occurring in the actual compilation process, the benchmark program was modeled along the lines of the actual compilation. A separate synthetic program is used to represent the make program, *gcc*, precompiler and compiler called *scomp*, *sgcc*, *scpp* and *scc1* respectively. The synthetic versions of assembler and linker programs were not created because we felt their basic nature in terms of system resource utilization was already captured by the synthetic precompiler and compiler. Calls to the assembler and linker are therefore replaced in the benchmark with calls to the compiler and precompiler respectively.

Each synthetic program is divided into two parts, one that exercises system resources and another that generates a proportionally correct amount of user time. The system resource code is a series of system calls as identified by profiling. Arguments for the system calls were chosen to generate the amount of utilization seen by the monitoring or files accessed. We provide the appropriate amount of user time by embedding the dhrystone benchmark code [WEI84] as a loop directly into each synthetic program and executing it in order to create the desired amount of user time.

Some Unix system calls identified by the profiling process were not available in all the operating systems that were to be tested. Most noticeably `vfork` was not available in the SCO System V/386. In order to maintain a common benchmark, we changed the `vfork` calls to `fork` in all the versions of this benchmark. All other system calls that were not common to all the operating systems did not contribute significantly to the execution time of the benchmark and were removed.

In order to isolate the performance of operating systems in providing specific types of system services used by the the compilation process, the benchmark can execute portions of the code related to either system utilization or user time. Within the system utilization code, execution can be limited to those system calls and activities associated with file operations, file reads and writes, memory operations and process management.

Time was measured in this program using `ftime`.

### 4.3 Jigsaw Puzzle Benchmark (*jigsaw*)

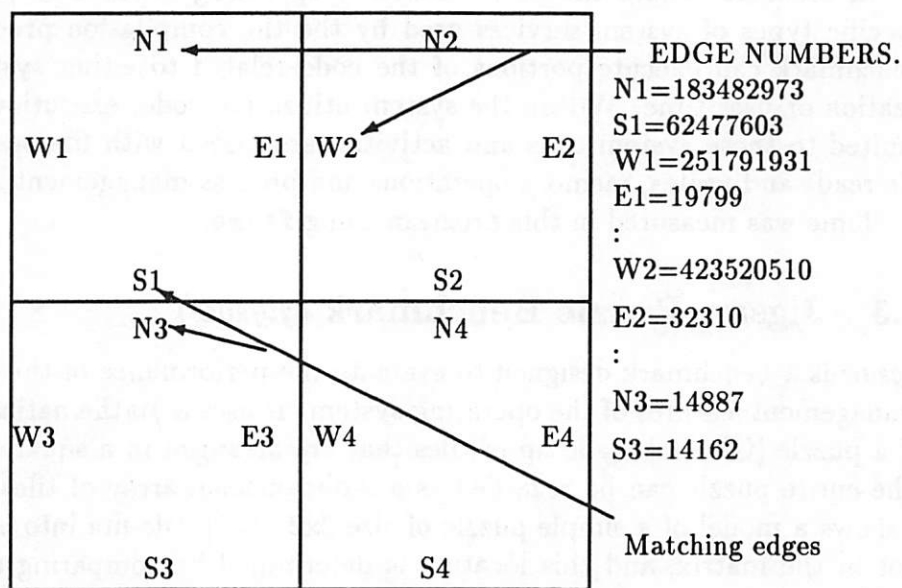
*jigsaw* is a benchmark designed to evaluate the performance of the memory management features of the operating system. It uses a mathematical model of a puzzle [GRE86] made up of tiles that are arranged in a square matrix. The entire puzzle can be regarded as a 2-dimensional array of tiles. Figure 1 shows a model of a simple puzzle of size 2x2. Each tile fits into a specific slot in the matrix, and this location is determined by comparing the edge numbers of the tile being considered, with those of its neighbors. The type of puzzle that is modeled is torroidal, where the left edge of the puzzle wraps around to the right edge and the bottom edge wraps around to the top edge.

#### 4.3.1 Overview

*jigsaw* has 3 stages of execution:

- The puzzle generation stage: During this stage, the program accepts the size of the puzzle to be generated and dynamically allocates the memory required for the puzzle, and makes each tile of the puzzle by assigning its edge numbers and identity. The tiles are then connected to form a linked-list.





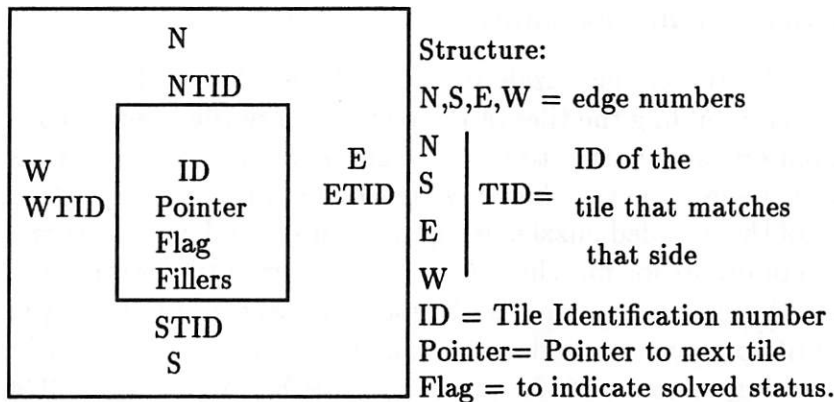
S1,N3 forms a simple pair of matching edges, while N1,S3 forms a torroidal match. Two edges match if the remainder of dividing the higher value with the lower equal a KEY value (101).

Figure 1: Mathematical model of a 2x2 puzzle

- The puzzle scrambling stage: The links of the tiles are re-arranged so that the sequence of the tiles gets jumbled. A fixed re-arrangement sequence is used in place of the suggested random sequence [GRE86], to ensure that the amount of computation for different runs are the same.
- The solution stage: The scrambled puzzle which is now in memory, is solved during this stage. In order to eliminate the effects of the delays caused by the secondary storage devices, only the solution process of the program is timed, using system calls `ftime` or `gettimeofday`, right before and after the solution stage.

#### 4.3.2 Structure of a tile

Figure 2 shows the structure of a tile in the puzzle.



Fillers are used to increase the memory requirement of the tile.

Figure 2: Structure of a tile

It is assumed that each tile has four sides. A tile is represented as a record with fields for numbers to represent the four sides. These numbers are all unique and are used for finding the side of another tile that matches with it. For the sake of simplicity, only the left side of any tile can match with the right side of another, south side of one with the north side of another and so on. Each tile also has a field to hold its identifier and four fields (one for each side), initially blank, to record the identifier of the matching tile to that

side found while solving the puzzle. In addition, the tile structure has fillers to increase the memory occupied by the tile. By adjusting the parameters of the puzzle, such as the number of tiles in the puzzle and the size of each tile, the amount of computation and the memory requirement can be varied.

#### 4.3.3 Solving the puzzle

The solution of the puzzle begins by picking the first unmatched tile in the sequence and comparing its edges with those of the remaining tiles until any side matches with any side of another tile. At this point, the IDs of the matching tiles are stored in the corresponding fields of the two tiles. When all four sides of the tile have been matched, the next tile is taken, to match the remaining sides. The solution continues until all tiles have been placed in the correct position.

#### 4.3.4 Paging activity during solution

The movement of the tile of the puzzle is a logical one. It happens in two parts. At first, when jumbling the tiles of the puzzle, every tile is accessed to reposition the pointers of each tile, to point to another random tile, instead of the next logical tile in sequence. Later, while solving puzzle, the program accesses each tile of the jumbled puzzle, using the pointers and compares each tile with the sides of others for matches. At this point, each tile, represented by at least a KB of memory or more, is accessed excessively for comparing sides, and four write operations are done on each tile. For example, in order to complete the solution for a 32x32 puzzle, *jigsaw* has to perform 4096 writes and an average over 2 million reads to tiles in memory. If each tile, represented by 4096 bytes, form a page, *jigsaw* would make over 2 million page requests!

## 5 Results of Operating System Comparison Tests.

### 5.1 Equipment Configurations

Our tests were run on Hewlett-Packard Vectra RS/25C Model 304e machines. These machines are connected to the campus wide Ethernet Local Area Network. They have been set up so that it is possible to run these machines on the campus network or connected together in a stand-alone network.

One of the machines is running the Mach 2.5 operating system (release i386m, July 10 1990), one is running Mach 3.0 (release #6.0, June 19/20, 1990), and the third is running Santa Cruz Operations (SCO) Unix System V/386, Release 3.2. To compile our benchmark programs, we used the *gcc* compiler on the Mach machines, and the *cc* compiler provided with the distribution on the SCO System V/386 machine.

Each of these machines has the following configuration:

- 25 MHz Intel 80386 CPU
- 310 Mbyte ESDI hard disk drive,
- 5.25inch/1.2 Mbyte floppy drive,
- Flexible disk and ESDI hard disk controller
- 16 Mbytes of RAM
- Enhanced HP Vectra PC keyboard
- High resolution Display controller card.
- 16" High resolution Monitor (1280x1024 pixels)
- HP 27210A ThinLAN Interface Card (Ethernet)
- Serial Port
- Parallel Port

## 5.2 Numerical Results and Discussion

### 5.2.1 Synthetic Dump Tests

Results of *sdump*, given in Table 1 shows that Mach 2.5 has the best performance, compared to SCO System V/386 and Mach 3.0. Tests were done for data sizes varying from 1MB to 14MB. The entire range of tests showed that Mach 2.5 performed approximately 6% better for lower data sizes and about 10% better for large data sizes than SCO on a 386 machine.

This program stresses the system in terms of memory requirement and the management of virtual memory. Mach 3.0 failed to handle data files that were larger than 5MB. The failure is believed to have taken place when an attempt is made to create a file that is greater than 5MB. Other programs showed similar effects on Mach 3.0. For example, a simple attempt to catenate multiple files into a large one, greater than 5MB failed.

The figures for the benchmarks were obtained after multiple runs of these programs. The standard pattern that was observed was that, the first run took about 10% more time to run than subsequent runs for smaller data sizes. This pattern was observed on all the test beds. However, for higher data sizes, this effect was insignificant. Successive executions yielded similar results, with less than 1% difference between them. The first 4 runs were recorded and their average are reported. This was observed due to file caching that happens when all of the data can be fit into the available memory. As memory requirement equals or exceeds the memory available, effects of file-caching are less felt.

Table 1: Synthetic Dump (*sdump*) Results

Av data size MBytes	No of files	Mach 2.5 KB/sec	Mach 3.0 KB/sec	SCO System V/386 KB/sec
4.38	3	175	52	164
7.14	5	174	*	158

\* Mach 3.0 crashes for large data sizes with an error:  
"Trap type 3, Bus Panic"



### 5.2.2 Synthetic Compilation Tests

The results of running the synthetic compilation program on the test systems are shown in Table 2. Results are reported for running the complete benchmark, the benchmark with only the system utilization code (no dhrystone loops to generate user time) and with only the code representing the specific system utilization areas of file operations, file reads and writes, memory management and process management. The code for process management is executed in all versions.

For the benchmark in general the results show that system utilization code makes up roughly 10% of the total execution time of the benchmark. Within the system utilization code the memory management section is responsible for approximately 65% of the execution time followed by the file reads and writes, file operations and process management sections with approximately 20%, 10% and 5% respectively.

In overall system utilization Mach 3.0 is faster than Mach 2.5 which is faster than the SCO System V/386. Mach 2.5 is faster than Mach 3.0 in all system utilization areas except memory operations. This area dominates the system utilization execution time of the benchmark and the performance difference is large enough to make Mach 3.0 the overall winner. The largest proportional difference between the operating systems is in process management calls where Mach 3.0 and SCO System V/386 are from twice to three times slower than Mach 2.5.

A comparison of the complete benchmark results with the system utilization results shows that user time accounts approximately 90% of the execution time of the benchmark (and of the actual compilation on which this is based). Thus we see that compilation is not an operating system intensive application.

Each test was run three times, and the figure reported in Table 2 is the average of the three runs. The times for the individual runs varied by less than 2% from each other.

### 5.2.3 Jigsaw Puzzle Tests

Results of execution of the *jigsaw* for two tile sizes are presented in Table 3.

This program stresses the machine in two ways:

Table 2: Synthetic Compilation (*scomp*) Results

Test Type	Mach 2.5 sec.	Mach 3.0 sec.	SCO System V/386 sec.
Complete	810.0	795.7	942.5
System Utilization Code	82.3	74.1	87.1
File Operations	12.0	15.0	14.2
File Reads and Writes	19.4	21.4	22.8
Memory Management	56.8	50.7	64.4
Process Management	2.7	4.8	7.0

1. It dynamically allocates large amounts of memory for the entire matrix.
2. It then requests pages of memory at a very rapid rate for reading and for writing.

The puzzle size is gradually increased to a stage at which the system fails. Each run was monitored on the Mach 2.5 system to see the paging patterns. This was done by using the Mach system call, *vm\_statistics*, before and after execution. This yielded among other figures, the total page faults and separate figures for soft and hard page faults. If a requested faulted page is to be obtained from the disk, it is regarded as a hard fault, else it is a soft fault.

Table 3 shows the test results run on different environments, up to failure. The performance of the virtual memory management was not as expected. It was observed that as long as the page faults that occurred were soft faults, the program's demands were met by the system. Hard faults were observed to begin for puzzle size 32x32, with 4MB tile size and sharply increased with higher puzzle sizes. The program execution failed, signaling a "Bus Error" on SCO system and "Error: Trap type 3, Bus Panic" on Mach systems, when trying to solve puzzles of large sizes. On the Mach systems, these programs were re-compiled with debug option and were run under the debugger *gdb* which reported that the crash occurred from within the kernel.

Mach 2.5 was observed to be the most stable of all the systems tested. The SCO System V/386 was very unstable and an unusual observation was that the program used to run for some values of memory requirements but not for others. The reason for this behavior is not known.

Mach 3.0 performed the best before it failed, when compared to Mach 2.5 and SCO, in most runs. Unfortunately, unlike Mach 2.5, it was not stable for higher puzzle sizes. In all cases, the failure was seen at the time of solving the puzzle, where pages are requested at a rapid rate. All the systems could handle the generation and jumbling of the puzzle, since paging was gradual in this part of the program.

The figures reported in Table 3 are the average of three runs of the *jigsaw*. It was observed that there was very little variation in the times reported by consecutive executions of the benchmark.

## 6 Conclusions and future work

We have presented benchmark programs which test system services in a variety of areas, and shown the results of running these tests on Mach 2.5, Mach 3.0, and SCO System V/386. At present we are working on extending these results in a number of directions.

First, we are constructing additional benchmarks. One will be based on the operation of the X window system, emphasizing interprocess communication and network traffic. In this effort we have been helped very much by some tests and performance tools presented in [DRO90]. We are also developing a benchmark based on the operation of a database system, which exhibits a very different sort of file access patterns than shown in our synthetic dump program.

We would also like to run our benchmarks on a wide variety of machines. This will require finding appropriate parameter settings for our benchmarks for different classes of machines.

In addition, we would like to run the tests concurrently to observe the performance of the system under load. This could be accomplished by running multiple copies of the same test, or different tests concurrently.

The issue of using benchmarks to compare incompatible systems also needs to be addressed. This necessitates leaving the domain of Unix-syntax benchmarks, and using the different system calls that each operating system permits. While it is obviously more difficult to ensure the fairness of such comparisons, this approach allows each operating system to use those features

Table 3: Jigsaw Puzzle (*jigsaw*) Results

Tile size KB	Puzzle Size	Mach 2.5 msecs.	Mach 3.0 msecs.	SCO System V/386 msecs.
1	8x8	30	20	-
	12x12	100	90	116
	16x16	230	230	-
	18x18	360	360	-
	24x24	975	930	1000
	32x32	2360	2470	-
	40x40	5590	-	-
	45x45	8620	-	-
	50x50	12260	-	-
	55x55	16690	-	-
	60x60	21660	-	-
	62x62	-	-	-
4	8x8	30	30	-
	12x12	105	100	124
	16x16	250	240	-
	18x18	370	-	-
	24x24	1003	-	1693
	32x32	2546	-	-
	40x40	5715	-	-
	45x45	-	-	-
	50x50	-	-	-
	55x55	-	-	-
	60x60	-	-	-

Note: A "-" indicates unsuccessful execution.

A blank space indicates that the test was not executed.

that allow it to work best.

## Acknowledgements

This research is supported by a grant from the Research Institute of the Open Software Foundation. The authors wish to thank ACE (Associated Computer Experts, bv, Amsterdam) for providing a copy of their benchmarking report. The authors also wish to thank Mr. Dhruve K. Shah of Worcester Polytechnic Institute, the co-author of the *jigsaw* puzzle program described in Section 4.3.

## References

- [SPE90] "Benchmark Results ", *SPEC Newsletter*, Vol. 2, No. 2, Spring 1990.
- [ACE89] *Benchmarking UNIX Systems* , ACE (Associated Computer Experts bv, Van Eeghenstraat 100, 1071 GL Amsterdam, The Netherlands, 1989.
- [BLA89] D.L. Black, R.F. Rashid, D.B. Golub, C.R. Hill, and R.V. Baron, "Translation Look-aside Buffer Consistency: A Software Approach", *Digest of Papers, COMPCON Spring '89, Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage* , (1989), 184 - 190. [ Also Carnegie Mellon University School of Computer Science Technical Report, CMU-CS-88-201].
- [CUR76] H.J. Curnow and B.A. Wichmann, "A Synthetic Benchmark", *The Computer Journal*, 19 (1976), 43 - 49.
- [DON87] J. Dongarra, J.L. Martin and J. Worlton, "Computer Benchmarking: Paths and Pitfalls", *IEEE Spectrum*, July 1987, 38 - 43 , .
- [DRO90] R.E. Droms and W.R. Dyksen, "Performance Measurements of the X Window System Communication Protocol", Bucknell University Computer Science Department Technical Report #90-9, March 1990.



- [FOR89] A. Forin, J. Barrera, M. Young, and R.F. Rashid, "Design, Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach", Carnegie Mellon University School of Computer Science Technical Report, CMU-CS-88-165. [Also published as "The Shared Memory Server", USENIX Winter Conference, San Diego, 1989.]
- [GOL90] D. Golub, R. Dean, A. Forin, and R. Rashid, "Unix as an Application Program", *Proceedings of the USENIX Summer Conference*, June, 1990, 87 - 95.
- [GRA82] S.L. Graham, P.B. Kessler, and M.K. McKusick, "gprof: A Call Graph Execution Profiler", *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 17, No. 6 (June 1982) 120 - 126.
- [GRE86] P.E. Green and R.J. Juels, "The Jigsaw Puzzle - A Distributed Performance Test", *Proceedings of the 6th International Conf. on Distributed Computing Systems*, May 19-23, 1986 288 - 295. .
- [SMI90] B. Smith, "The Byte Unix Benchmarks", *Byte*, March 1990, 273 - 277.
- [TEV87] A. Tevanian, "Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach", Ph.D. Thesis, Carnegie Mellon University School of Computer Science, Dec., 1987. [ Also Carnegie Mellon University School of Computer Science Technical Report, CMU-CS-88-106].
- [WEI84] R.P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark", *Comm. of the ACM*, 27 (1984), 1013 - 1030.

# A Revised IPC Interface

Richard Draves\*

## 1 Introduction

The Mach 3.0 IPC facility efficiently supports many different styles of communication, including server-client remote procedure calls, distributed object-oriented programming, and streams. As a building block, the interface provides a general message primitive. The message primitive operates on capabilities for communication ports. The interface remedies numerous problems that were observed in the older Mach 2.5 interface. The implementation provides good performance, running 30-40% faster than Mach 2.5 with data structures 50% smaller. The Mach 3.0 implementation provides backwards-compatibility support for the older interface.

## 2 Interface Description

The Mach IPC facility provides message-oriented, capability-based interprocess communication. The interface supports several different styles of interaction, including remote procedure calls, object-oriented distributed programming, and streams. The implementation makes use of the VM system to efficiently transfer large amounts of data using a copy-on-write optimization. The Mach IPC interface allows intermediaries to transparently extend the IPC facility. A user-level server extends Mach IPC across networks.

### 2.1 Remote Procedure Calls

The IPC interface supports remote procedure calls as an important special case operation. The `mach_msg` system call allows tasks to combine send and receive operations. The interface provides send-once rights for ports, designed for use as reply port rights. A notification mechanism allows clients

---

\*This research supported in part by the Fannie and John Hertz Foundation.

and servers to detect and recover from aborted RPCs. The MiG stub generator aids the construction of RPC interfaces.

At the message level, an RPC consists of a request message sent from a client task to a server task and then a reply message sent from the server task to the client. The server task holds a receive right for the service port, and the client task holds a send right for the port. Many client tasks may have send rights for a single port. The `mach_msg` system call satisfies the needs of both client and server tasks with a single system call. Client tasks use `mach_msg` to send a request message and receive the reply. Server tasks use `mach_msg` to send a reply message and receive the next request.

The interface provides send-once rights, for use as reply ports. The client provides the server with a send-once right for a reply port, to which the server sends the reply. The request message carries the send-once right. The message header contains a field for transmitting rights to a reply port. The holder of a send-once right can use it to send only one message, because the act of sending to a send-once right deletes it. When the client receives its reply message, it knows that the server retains no rights for the reply port.

Send-once rights never disappear; they always result in a message. When a task deallocates a send-once right, the kernel uses the send-once right to send a notification message. If a server dies while processing an RPC, the server's client can recover because it receives this notification message from the kernel.

The interface also provides dead names and dead-name notifications, which allow servers to detect the death of clients and abort computations. When a client dies, the kernel deallocates its port rights, including its reply port. The send-once right for the reply port, held by a server, becomes invalid and turns into a dead name. When the server tries to send its reply message to the dead name, the `mach_msg` call returns an error that informs the server of its client's death. If a server wishes to abort computations on behalf of dead clients, it can request dead-name notifications for the send-once right reply ports. Then when a client's send-once right becomes a dead name, the kernel will send a dead-name notification message to the server, informing it of the new dead name.

Mach provides an RPC stub generator, MiG, which simplifies the construction of distributed systems by implementing familiar procedure-call functionality. For the client, MiG produces stub procedures which pack a request message, use `mach_msg` to send the request and receive a reply message, and then unpack the reply. For the server, MiG produces stub

procedures which unpack the request message, call the server's function, and then pack a reply message. MiG also provides servers with a standard function to receive requests and send replies using `mach_msg`. MiG allows clients to send asynchronous messages when they do not need a reply message.

## 2.2 Object-Oriented Servers

Mach IPC supports the object-oriented construction of distributed systems. In this model, a port represents an object, and messages sent to the port manipulate the object. Send rights for the object port act as opaque object pointers. Port sets let a few threads serve requests for many objects. A no-senders mechanism allows object servers to garbage-collect unused objects. The MachObjects package simplifies the construction of object-oriented interfaces. The kernel interface represents a good example of an object-oriented interface.

The interface allows object-oriented clients and servers to use send rights as object pointers. Programs can compare send rights for equality because a task only holds one send right for any given port. If a task holds a send right for a port, and receives another send right for the same port in a message, then the kernel reuses the task's existing name for the send right.

Port sets let an object-oriented server manage hundreds or thousands of objects. A port set holds receive rights, and a receive operation on the port set returns the next message sent to any of the constituent ports. The received message specifies the port to which it was sent, so that the server knows what object to manipulate. A server with many objects must rely on port sets, because assigning a separate thread to every object would consume a prohibitive amount of space for stacks and other data structures.

The interface provides a no-senders detection mechanism. This mechanism allows the holder of a receive right to detect the absence of send rights for the receive right, and garbage-collect the receive right and the object it represents. The task holding a receive right can request a no-senders notification for the right, supplying a send-once right to which the kernel will direct the notification. Then when the last send right for the port disappears, the kernel uses the send-once right to send the task a no-senders notification.

The no-senders detection applies only to send rights, not send-once rights. The receive right holder does not need a notification to detect an absence of send-once rights for a port. Because only the receive right holder

can create new send-once rights, and send-once rights do not disappear but always result in a message, the holder of a receive right can keep track of the number of send-once rights without further help from the kernel.

Because send-once rights do not affect no-senders detection, the no-senders notification for a port can be sent to the port itself. This significantly reduces the number of ports used by object-oriented servers, because they do not need to create a notify port for every object port. Clients holding send rights for the object port can not forge no-senders notifications, because the server can distinguish messages sent to send-once rights from messages sent to send rights.

The MachObjects package aids the construction of complex object-oriented interfaces. MachObjects provides functionality similar to Objective C. It allows the definition of classes, the inheritance of methods for handling request messages, and the delegation of operations to another object. Like MiG, it packs and unpacks messages for the application.

The kernel interface demonstrates object-oriented principles. Ports represent kernel objects like tasks and threads, and user programs can manipulate any task or thread for which they have send rights with RPC operations on the ports. This approach lets applications like the Mach 3.0 Unix emulation and debuggers manipulate other tasks, even tasks on other machines, through the standard kernel interface.

## 2.3 Streams

Because the IPC interface provides message-oriented communication, it directly supports stream protocols. The interface guarantees reliable message delivery. The interface also supplies flow control mechanisms for both readers and writers.

The IPC interface requires reliable, sequenced message delivery. The implementation must deliver every message sent to a port, unless the port is destroyed. Furthermore, the implementation must preserve the order of messages from a single source. The interface does not define the ordering of messages sent by different threads. Network message servers extend these semantic guarantees across networks.

A stream reader adjusts the message queue's length to perform flow control. The reader holds a receive right for the port implementing the stream. Queue of messages at the port may only grow to a limited size, controlled with the `mach_port_set_qlimit` call. By modifying the queue limit the reader controls the buffering in the stream.



The interface also offers stream writers a flow control mechanism. Normally when the writer exceeds the port's queue limit, the writer's send operation blocks until the reader reduces the size of the queue. Alternatively, the writer may request a msg-accepted notification. In this case, the kernel sends the writer a notification message to inform the writer that it can resume queuing messages. This option allows the writer to perform other work while waiting for the reader to catch up.

## 2.4 Copy-On-Write

The IPC implementation makes use of the VM system to efficiently transfer large amounts of data. A VM optimization, copy-on-write sharing, lazy-evaluates the copy operation. The implementation also allows a message sender to move data into a message, further reducing the transmission cost.

A message body can contain the address of a region in the sender's address space which should be transferred as part of the message, as an out-of-line memory region. When a task receives a message containing an out-of-line region, the region appears in an unused portion of the receiver's address space. Out-of-line data retains the normal copy semantics of message transmission. The sender and receiver can freely modify the out-of-line data. The implementation optimizes the transmission of out-of-line data so that sender and receiver share the physical pages of data copy-on-write, and no actual data copy occurs unless the sender or receiver modifies the out-of-line data. Regions of memory up to the size of a full address space may be sent in this manner.

A message sender can enable a further optimization when it does not wish to retain a copy of the memory region. The deallocate bit in the out-of-line data's type descriptor indicates that the memory region should be deallocated from the sending task. The VM system takes advantage of this information to disable copy-on-write, avoiding the cost of write-protecting the memory region. Instead of copying, the VM system moves the memory region directly from the sending task into the message.

## 2.5 Transparent Extensions

The IPC interface allows intermediate servers to transparently extend the IPC facility. User-level servers extend Mach IPC across networks; the Mach kernel only directly handles message transmission within a single machine. Programs can interpose on port rights, to transparently intercept message traffic.

Because the sender and the receiver of a message can not detect the presence of intermediaries, the IPC interface is transparent. Port rights carry no location information, so the sender of a message knows nothing about the true receiver. The interface does not give the receiver of a message the identity of the sender. The receiver only knows that the sender possesses a send or send-once right, because port rights are secure.

The user-level extension servers (network message servers, or netmsgservers) take advantage of the transparency and cooperate to forward messages across a network. When a remote task has send rights for a local port, it actually holds a send right for a proxy port on the remote machine. The remote netmsgserver holds the receive right for the proxy port, and the local netmsgserver holds a send right for the local port. When the remote task sends a message to its send right, the remote kernel delivers the message to the remote netmsgserver. The netmsgservers transfer the data over the network, and the local netmsgserver forwards the message to the local port using its send right.

The netmsgserver provides a simple name service which bootstraps this process. User tasks can register send rights with the local netmsgserver. The name service provides a lookup operation that a task can use to acquire send rights that were registered with netmsgservers elsewhere in the network. The lookup can be directed to a particular machine, or broadcast on the local network. The netmsgservers create proxy ports as needed to represent ports elsewhere in the network.

Netmsgservers can use encryption and authentication techniques to protect port capabilities across the network. They can encrypt message data. They make use of the type data in messages to convert between data formats. Implementation choices like level of security and network protocol are made at user-level, and a machine may support multiple extension servers offering different services.

Another application of transparency allows programs to intercept messages. For example, a debugger can monitor all message traffic sent through a send right. The debugger extracts the task's send right and inserts a send right for another port, owned by the debugger, using the `mach_port_extract_right` and `mach_port_insert_right` calls. Unbeknownst to the task, the debugger receives the messages sent by the task and forwards the messages to the extracted send right. Suspect programs may be run in a virtual mode, which intercepts for inspection all message traffic into and out of the task.

## 3 Interface Advantages

The Mach 3.0 IPC facility remedies several problems that were encountered when using the older Mach 2.5 interface to build complex multi-threaded applications. The old interface does not give complex applications the tools they need to manage their port rights correctly. Clients and servers written to the old interface face problems managing their reply ports. The old interface's design made it difficult for programs to use notifications effectively. Finally, the old interface does not allow programs to recover from some classes of errors. The new IPC interface overcomes these problems without unduly burdening the application programmer.

### 3.1 Port Right Management

The asynchronous nature of multi-threaded programs greatly complicates port right management. Multi-threaded Mach 2.5 applications commonly contain many subtle bugs because the old interface makes it difficult or impossible for these programs to manage their port rights correctly in the presence of port deallocation. Features introduced in the new interface, reference counts for port rights and dead names, allow multi-threaded programs to overcome these problems.

#### 3.1.1 Reference Counts

Reference counts for send rights allow programs to deallocate send rights safely. For example, consider the operation of a typical client program. The client looks up a service port, acquires a send right for the port, uses the send right to make requests, and deallocates the send right. If several threads in the client use the service concurrently, then the reference count for the send right prevents the threads from incorrectly releasing the send right. When a second thread acquires the send right, the right's reference count is incremented. When a thread deallocates the send right, the reference count is decremented. The client task only loses the send right when the last thread deallocates its reference for the send right.

A library can not solve this problem for Mach 2.5 programs by maintaining user-level reference counts for send rights. Even assuming the mandatory use of a standard library that provides such a coordinating facility, the library may not have complete knowledge of the references for send rights. A task may contain separate binaries, each with its own libraries and data

structures attempting to provide reference counts. The Mach 3.0 Unix emulation provides an example of this. The transparent emulation library is a separate binary loaded into the address space of Unix processes. The program and the transparent library can not coordinate their use of send rights because they do not share any data structures or code.

Even without separate binaries in a single task, a potential race condition prevents a standard library from coordinating send right references. A thread in the client task that receives a send right in a message immediately uses the library to register a reference for the send right. However, this leaves a window between when the thread receives the send right and when it registers it. In this window, another thread may release a reference for the send right. If it releases the last reference and the library deallocates the send right, then the first thread is deprived of the send right which it had just received.

The coordinating library avoid this race condition if it ensures that no receive operation is in progress when it deallocates a send right, perhaps by postponing deallocations. This solution eliminates the window, but it severely constrains the program. In many multi-threaded programs, most threads block in receive operations, so the coordinating library will never be able to deallocate send rights. In practice, most Mach 2.5 programs dodge the problem and never deallocate send rights.

### 3.1.2 Dead Names

Dead names give programs time to recognize the asynchronous deletion of a port right. When a port is destroyed, because its receive right is deallocated, the port's send rights become invalid. The new interface changes the invalid send rights into dead names. When a task tries to send a message to a dead name, the resulting error lets the task know that the port died and its send right is no longer valid. The task can then deallocate the dead name.

Like send rights, dead names have reference counts so that multi-threaded programs can safely manipulate them. When a send right with multiple references becomes invalid, the resulting dead name assumes the send right's references. As different components of the application notice the dead name and deallocate it, the dead name loses references. When the dead name loses its last reference the name becomes available for reuse by another port right.

A multi-threaded Mach 2.5 program is vulnerable to port name reuse when a port right becomes invalid. For example, after a thread in a client

task looks up a service port and receives a send right for the port, the port may die. The Mach 2.5 implementation deletes port's invalid send rights immediately. Another thread in the client task can create or receive a port right which happens to reuse the name of the invalid send right. Then the first thread will inadvertently use the wrong port right when it tries to make a request. In practice, this problem does not afflict old programs because the Mach 2.5 implementation allocates port names sequentially, so that names are very rarely reused. An implementation that picked the smallest name not currently used would break many Mach 2.5 programs.

### 3.2 Reply Port Management

Mach 2.5 clients and servers face several reply port management problems stemming from the old interface's use of send rights for reply ports. RPCs do not leave a client failure-isolated from the server. Servers must receive and discard many unwanted port-deleted notifications. Finally, clients can not recover when a server dies during an RPC. The new interface provides send-once right reply ports, which solve these problems.

The Mach 2.5 RPC model does not leave clients isolated from bugs or faults in the server after an RPC finishes. A faulty server can precipitate mysterious failures in any client which has used it. This negates some of the failure-isolation advantages gained by putting the client and server in separate address spaces. Because the client gives the server a send right for the reply port, the server can potentially send many messages to the reply port. Normally, the server sends only the reply message and then retains the reply port as an unused send right. However, clients can not count on this behavior. A buggy server may send additional messages to a reply port, and these messages will interfere with later RPCs made using the reply port. The only recourse for the client is to use a separate pool of reply ports, with one reply port per thread, for every service that the client uses. This alternative is prohibitively expensive, because it greatly increases the number of ports consumed by the client and because selecting the correct reply port for an RPC increases the RPC cost.

From a Mach 2.5 server's viewpoint, the reply port send rights it retains clutter its port name space. After a server sends its reply, it can not deallocate the send right reply port, because this exposes it to the previously discussed port deallocation problems. Even if a server could safely deallocate the reply port right, the deallocation operation would make RPCs significantly more expensive. When a client program exits, its reply ports are



deallocated. Every server which a client thread has used receives a port-deleted notification, which the servers must receive. The death of a client precipitates a flurry of context-switches as servers wake up to receive and throw away the port-deleted notifications.

When a Mach 2.5 server dies while handling an RPC, the client remains blocked waiting for a reply message that will never come. The client can not recover from the server's death. A time-out is not an appropriate solution to this problem, because very often the client can not pick an appropriate time-out value.

The new IPC interface provides a reply port field in messages, which carries send-once rights, to support RPCs. A client program wishing to make RPCs allocates a reply port, for which it holds the receive right. To perform an RPC, the client sends a request message to the server, using a send right for a service port held by the server. The request carries a send-once right for the reply port. The server sends its reply message to the send-once right, removing the send-once right from the server's port name space, and the client program receives the reply using its receive right for the reply port. If a client dies during an RPC, the server's send-once right turns into a dead name. When the server tries to send the reply, an error return notifies it that its client has died. If a server wishes to receive a more timely notification of the death of a client, the server can request a dead-name notification for the client's send-once right reply port. If a server dies during an RPC, the client receives a send-once notification for the reply port, so it unblocks and can recover from the failure.

### 3.3 Notifications

The new IPC interface's strategy of only generating notifications after an explicit request from a user task solves several problems with the Mach 2.5 notification model. The old model makes it impossible for different program components to share notifications. Most notifications generated by the Mach 2.5 kernel represent wasted effort because tasks do not want the notifications. Finally, the asynchronous generation of notifications makes it difficult for tasks to handle the notifications that it does want.

The new interface only sends notifications upon request. The notification request specifies a send-once right for the notify port. The kernel sends the notification to the supplied send-once right.

Because the old interface obliges the kernel to send port-deleted notifications for a dead port's every extant send right, the kernel often generates

unwanted notifications. Tasks must receive these notifications, so the death of a port triggers unnecessary context-switches and receive operations. The new interface avoids this problem, because tasks only receive notifications that they have requested.

The old interface directs all notifications to a single task-wide notify port, so separate program components or libraries that try to receive notifications suffer from contention. A standard user-level library to distribute notifications can not solve this problem, because a task may contain multiple binaries which wish to receive notifications. The new interface avoids this problem, because program components can specify send-once rights for different notify ports when they request notifications.

The old interface generates notifications spontaneously. If a task receives a send right for a port, and the port dies, the kernel immediately generates a notification. The task may receive and process the notification before it processes the received message that contains the dead send right. Because the task has not examined the send right, the port-deleted notification does not make sense to the task. The new interface avoids this problem, because the kernel will not generate a notification until the task examines the received message and explicitly requests a notification for the send right.

### 3.4 Error Recovery

The old IPC interface suffers from poorly defined semantics for many exceptional conditions. Mach 2.5 programs can not recover from some interrupted send operations, which may destroy critical port rights and out-of-line memory. Resources shortages during an old `msg_receive` system call may result in the destruction of ports and data, with no error indication given to the receiving task.

The old interface pretended that the send and receive message operations are atomic. Unfortunately, the Mach 2.5 implementation does not make send and receive atomic operations. Mach messages are complex objects, holding an unbounded number of port rights and out-of-line memory regions. It is very difficult, if not impossible, to make send and receive atomic. The new interface recognizes this and copes with the possibility of partially completed operations.

#### 3.4.1 Interrupted Send

Send operations can not always queue a message containing resources like port rights or out-of-line memory. When this happens, the new interface

returns the resources in the message to the sending task. If the task wishes, it can retry the send operation. The old interface destroyed the resources in the message.

The send operation has two phases. First, the kernel copies the message into an internal buffer, translating port rights and making out-of-line memory regions copy-on-write. Second, the kernel tries to queue the message. If the message can not be eventually queued, because the send times out or is interrupted, then the kernel must dispose of the message. However, the message may contain precious resources like a receive right or the sole copy of an out-of-line memory region.

The new interface specifies that an incomplete send operation returns the unsent message to the sender, giving the sender the port rights and out-of-line memory in the message. The sender can then retry the transmission. This return operation resembles a normal receive operation, and is known as a pseudo-receive.

The Mach 2.5 interface makes no provision for returning the contents of the message to the sender in these situations. Because it destroys the message, resources which are only present in the message may be irretrievably lost. For example, receive rights carried in the message are deallocated, destroying the ports. If out-of-line memory is sent with the deallocate option, then the sender does not retain a copy of the memory. This possibility means that if the send operation might block, the old `msg_send` system call is inherently unreliable when used to send receive rights or out-of-line memory with the deallocate option.

### 3.4.2 Partial Receive

Because a receive operation consumes resources, sometimes a resource shortage might prevent the reception of some resources carried in a message. The old interface does not allow programs to recover from this possibility, because the old `msg_receive` system call returns no error indication. The new interface specifies a simple semantics for partial receive operations that allows programs to perform error recovery.

When a task receives a message, the kernel transfers to the task the resources, port rights and memory regions, held in the messages. This process consumes resources. A port right may require a new name drawn from the task's port name space, and out-of-line memory requires an unused region in the task's address space. In addition, the kernel may need to allocate internal data structures. Because the receive operation is not atomic, an

allocation may fail after some resources have been transferred.

The Mach 2.5 interface does not specify the outcome of the receive operation when a resource allocation fails. In practice, the receive operation silently deallocates any troublesome port rights or out-of-line memory. The receiver gets no indication of a problem with the receive operation.

When some port rights or memory regions can not be received due to resource shortages, the new interface specifies the semantics of a partial receive. The partial receive attempts to give the receiver all the resources in the message. The partial receive destroys those resources that can not be transferred. The transfer of out-of-line memory containing port rights guarantees that the receiver never gets the port rights but not the memory. A task never receives a resource that it is not told about in the received message. The partial receive returns an error code that indicates what type of resources were lost.

## 4 Implementation

The Mach 3.0 implementation takes advantage of several properties of the IPC interface to provide good performance. The implementation uses highly tuned data structures that represent a port in 64 bytes and a port right in 16 bytes. At the same time, the data structures expedite important operations like looking up port rights given a task's name, creating port rights, and deleting port rights. When possible, the IPC code makes use of a scheduling optimization, hand-off scheduling, to reduce the context-switch overhead. In combination these optimizations make the implementation faster and smaller than the Mach 2.5 IPC implementation.

### 4.1 Data structures

The Mach 3.0 implementation represents port rights with a hybrid data structure. Most port rights fit into a per-task table, at 16 bytes per port right. The name of the port right is its index into the table. Because a task can rename its port rights to create a sparse port name space, the implementation must accommodate port rights that do not fall into the task's table. A per-task splay tree, a self-adjusting binary tree, represents these overflow port rights in 32 bytes. The kernel dynamically adjusts the division between the table and splay tree, growing the table as needed, to minimize memory consumption.

The new implementation solves the reverse translation problem, finding a name given a task and a port, in several different ways. Every port contains a pointer to the task holding the receive right and that task's name for the receive right. If that check fails, the kernel looks for a send right in the task's table of port rights. A closed hash table that translates send rights to port names is folded into the table of port rights. Finally, the kernel checks a global open hash table to find send rights represented in a splay tree.

This data structure takes advantage of two properties of dead names. First, dead names allow tasks to cope with an implementation that reuses names quickly. The per-task tables contain a free list of unused entries. The entry for a deleted port right is pushed onto the free list, and it will be reused for the next port right to be allocated. Second, dead names allow the implementation to lazy-evaluate port destruction. The data structure does not allow the kernel to traverse a dead port's send rights and convert them to dead names. Instead, the kernel converts the send rights to dead names as it comes across them. The kernel uses a separate list of send-once rights attached to the port to generate dead-name notifications.

The hybrid data structures makes the important operations efficient. A name lookup, to convert from a task's name for a port to an internal port pointer, performs a bounds check and indexes into the task's table. The allocation of a send-once right pops an unused entry off of the free list, and the deallocation of the send-once right pushes the entry back onto the free list. Because every send-once right gets its own port name, send-once rights are not entered into the reverse hash tables. A simple RPC with a send-once right reply port does not use the reverse translation algorithm.

The Mach 2.5 implementation uses a straightforward data structure. It represents every port right with a 52-byte structure, and every port with an 88-byte structure. Each port right is on four doubly-linked lists. Each port has a list of its rights, each task holds a list of port rights, and each port right belongs to two global open hash tables. The hash tables perform the translations from task and name to port and the reverse translation from task and port to name. This representation of port rights makes all operations possible, but no operations are particularly efficient.

## 4.2 Hand-off scheduling

The hand-off scheduling optimization significantly reduces the cost of an RPC. Hand-off scheduling directly transfers control of the processor from one thread to another. The new IPC interface allows the Mach 3.0 imple-



mentation to take full advantage of hand-off scheduling.

The `mach_msg` system call allows the implementation to do hand-off scheduling in both directions of an RPC. A fast RPC implementation makes two hand-offs. First the client threads hands-off to a server thread blocked doing a receive, leaving itself blocked waiting for the reply. Then the server thread hands-off to the client, leaving itself blocked waiting for the next request. If both client and server use the `mach_msg` call to combine the send and receive operations, the implementation will avoid queuing and dequeuing of the messages and instead hand messages and control of the processor directly from sender to receiver.

The Mach 2.5 implementation only achieves a hand-off in one direction, from client thread to server thread. The old interface's `msg_rpc` system call, used by the client, does not provide enough flexibility. `msg_rpc` first sends to the destination port a message that carries a reply port, and then it receives from the reply port. Servers can not use `msg_rpc` to send a reply message and then receive a request, because reply messages should not carry a reply port and `msg_rpc` can not receive from a port set.

### 4.3 Performance

Space and time are both important performance measures. The Mach 3.0 Unix emulation relies heavily on the Mach IPC facility. A current single-server Unix system uses approximately 2000 ports. The emulation of many Unix system calls requires one or more Mach RPCs. The multi-server system under development makes even greater use of the IPC system.

The following table presents performance numbers for a DECstation 3100 running a Mach 3.0 kernel, version XMK23, with single-server Unix emulation, version XUX16.

### 4.4 Implementation History

The implementation of the new interface went through several phases, culminating in the current Mach 3.0 implementation. The goal was to test the new interface with a prototype before putting substantial time into the radically different implementation that was envisioned.

A modified version of Mach 2.5 first implemented the new interface. The augmented Mach 2.5 data structures did not perform well, principally because the creation and deletion of send-once rights was relatively slow. However, the prototype implementation did allow test programs to exercise

		Mach 3.0	Mach 2.5
kernel ports	976		
user ports	1320		
total ports	2296	146944 bytes	202048 bytes
kernel rights	976	0 bytes	50752 bytes
rights in tables	2447		127244 bytes
total size of tables	3032	48512 bytes	
rights in trees	37	1184 bytes	1924 bytes
total size		196640 bytes	381968 bytes
null RPC		125 $\mu$ s	210 $\mu$ s

Figure 1: Performance Comparison

the new interface. The test programs demonstrated the usefulness of the new features. This evaluation was used to refine the new interface.

The core of the new implementation, about 80% of the total including all the data structures, was developed and tested as a user-level library. First, interactive programs tested the low-level data structures. After the higher-level message functions and kernel calls were written, they were tested with communicating user-level coroutine threads. Finally, the code was moved into kernel space and mated with the Mach kernel. This development process produced an exceptionally trouble-free implementation.

The Mach 2.5 netmsgserver runs under the new interface, using the compatibility code. Although new clients and servers can use the netmsgserver to communicate over the network, the performance is oppressively poor. Because the netmsgserver sees the send-once right reply ports as send rights which are destroyed, after every RPC it goes through a broadcast protocol to destroy the reply port. With knowledge of send-once rights, a revised netmsgserver can avoid this performance pessimization.

## 5 Compatibility

The Mach 3.0 IPC facility includes complete backwards-compatible support for the older Mach 2.5 interface. This design requirement motivated some aspects of the Mach 3.0 interface. Programs written to the older interface make use of a compatibility mode that supports the older system calls and

kernel calls. The compatibility mode allows programs to use both interfaces simultaneously. Programs written to the new and old interfaces can even interoperate, exchanging messages and forming client-server relationships. The compatibility mode does not affect the performance of programs written to the new interface, and if desired the Mach 3.0 kernel can be compiled without the compatibility code.

## 5.1 Compatibility Requirements

The compatibility requirement motivated some important design decisions for the new interface. If it were not for compatibility considerations, the IPC interface could be potentially have simplified message formats and port right management.

The interoperability requirement prevented radical changes to the message format. The Mach 3.0 message format, a message header followed by typed data, differs from the Mach 2.5 message format in only minor details. Because the formats are so similar, programs using the compatibility mode can interoperate with programs written to the new interface. The implementation converts between the two formats when a programs uses the old system calls to send and receive messages.

The new interface allows reply ports to be send rights or send-once rights. It would be simpler if only send-once right reply ports were allowed, but this would make it difficult to convert some programs to the new interface. These programs use the reply port field as a general mechanism for conveying a send right, instead of restricting its use to RPCs.

Compatibility considerations motivated the design of reference counts for send rights. An alternative design, without reference counts, would make send rights behave like send-once rights. When a task would receive a port right, the port right would always appear in the task's port name space with a new name. A task could potentially hold many send rights for a single port. This alternative would simplify the interface and the implementation, but it would prohibit backwards-compatibility. Some programs rely on the current behavior to perform authentication and naming functions by comparing send rights.

## 5.2 Compatibility Support

The compatibility code supports the complete Mach 2.5 interface, with no exceptions. This does not mean that it is impossible to write a program which will determine under which implementation it is running. However,

it does guarantee that correct programs will continue to operate. The compatibility code hides new features like send-once rights from old programs.

The compatibility code supports programs which use both the old and new IPC interfaces. This allows a program to be gradually converted from the old interface to the new interface. For example, libraries may be written to different interfaces. This occurs frequently in practice, because the Mach 3.0 transparent emulation library uses the new IPC interface. Any process using the old interface has the emulation library in its address space, using the new interface to implement the Unix emulation.

Although the new and old interfaces are similar in many respects, they share no names. The new interface prefixes every name with "mach\_" or "MACH\_". This prevents name conflicts between the interfaces. It allows a common set of include files to serve both interfaces, and allows a binary to call freely functions from both interfaces.

The compatibility mode operates on a per-port right basis, not per-task. For example, a send right received with the old `msg_receive` system call is marked internally with a compatibility flag. If the port is destroyed, the send right disappears and the kernel sends a port-deleted notification to the task's notify port, implementing the old semantics for port destruction. The same task may also hold other send rights received with the new `mach_msg` system call. When those ports are destroyed, the send rights turn into dead names.

The compatibility code disguises send-once rights as send rights. When a task receives a send-once right with an old system call, like `msg_receive`, the task gets what looks like a send right. However, when the task sends a message to the faux send right, the right disappears and the task receives a port-deleted notification. This behavior preserves the essential send-once character of the right. It allows servers written to the old interface to handle clients using send-once right reply ports.

### 5.3 Implementation Impact

The compatibility code does not affect the Mach 3.0 interface's performance. The code does slightly increase the size of the kernel, but a build-time option allows the code to be removed if this is a problem. In many respects, the Mach 3.0 compatibility code is a better implementation of the Mach 2.5 interface than the Mach 2.5 implementation, because the compatibility code does not suffer from the old implementation's bugs. However, the compatibility code does perform more slowly than the Mach 2.5 implementation.

The new implementation does not optimize the common paths through the compatibility code.

## 6 Conclusion

The Mach 3.0 IPC facility revises the Mach IPC interface to improve support for multi-threaded programs, remote procedure calls, and object-oriented programming. The new interface corrects several problems in the older interface that made it difficult to use correctly. The Mach 3.0 implementation meets the needs of demanding applications, like the Mach 3.0 Unix emulation, with substantially improved space and time performance. Finally, the implementation supports the old Mach 2.5 interface with a compatibility mode that lets programs gradually migrate to the new interface.

## A Interface Summary

### A.1 System calls

`mach_msg(msg, options, send_size, rcv_size, rcv_name, timeout, notify)`

Send a message, receive a message, or do both. When sending, the message header contains a right for the destination port. Some options use the timeout value and the notify port.

`mach_host_self()` Returns a send right for a port representing the caller's host.

`mach_task_self()` Returns a send right for a port representing the caller's task.

`mach_thread_self()` Returns a send right for a port representing the caller's thread.

`mach_reply_port()` Allocates a port in the calling task and returns the name of the receive right. This call provides a way of allocating a reply port that does not use a reply port. The implementation may optimize the allocated port for use as a reply port.



## B Kernel calls

`mach_port_allocate(task, what_right, OUT name)` Allocates a port, port set, or dead name in the task. Returns the name of the new right.

`mach_port_allocate_name(task, what_right, name)` Allocates a port, port set, or dead name in the task. The new right has the specified name.

`mach_port_get_refs(task, name, what_right, OUT refs)` Returns the number of references held by the task for the specified port right.

`mach_port_mod_refs(task, name, what_right, delta)` Modifies the number of references held by the task for the specified port right.

`mach_port_deallocate(task, name)` Deallocates a reference for a send right, send-once right, or dead name.

`mach_port_destroy(task, name)` Deallocates all references and rights denoted by the name.

`mach_port_type(task, name, OUT type)` Returns the type of the specified port right.

`mach_port_names(task, OUT names, OUT types)` Returns a list of the names in the task's port name space, with a corresponding list of port right types.

`mach_port_rename(task, old_name, new_name)` Renames a port right.

`mach_port_get_receive_status(task, name, OUT status)` Returns status information for a receive right.

`mach_port_set_qlimit(task, name, qlimit)` Modifies a receive right's queue limit.

`mach_port_set_mscount(task, name, mscount)` Modifies a receive right's make-send count.

`mach_port_move_member(task, port_name, set_name)` Moves a receive right into or out of a port set.

`mach_port_get_set_status(task, set_name, OUT members)` Returns a list of the members of a port set.

The Napier88 programmer has no control of the store — the underlying system manages the creation, movement and garbage collection of all objects. Protection is provided by the type system and all Napier processes reside in the same large (conceptually infinite) persistent address space. The robustness of the system lies in its ability to stabilise. Stabilisation, which may be invoked by the user, causes a backup of the system's state to non-volatile storage. Any subsequent failure causes the system to return to the last stabilised state.

The architecture described in this paper is partially derived from a layered storage architecture developed by Brown [5]. This architecture has the potential to support a variety of persistent programming languages, such as PS-algol [6] and Galileo [7], as well as being sufficiently flexible to be used as an experimental platform. There have been many experimental systems constructed to support persistent programming languages [8,9,10]. We have chosen to use the well-known model consisting of a central object repository (server) and distributed computing agents (clients). Napier88 programs are executed by clients which may run on the same machine as the server, or on machines connected to the server. Clients do not contain any stable persistent storage; this is provided by the central Stable Store Server.

## **2. Experimental environment**

### **2.1. Napier88**

The Napier88 language was developed by the PISA project [11] as a test-bed for experiments in type systems, programming environments, concurrency, bulk data objects and persistence. The Napier88 type system is polymorphic and evolved at the same time as Cardelli and Wegner [12] published their work. Many of the ideas are related to theirs and some have been borrowed from them. The philosophy is that types are sets of values from the value space. The possibility of static type checking is retained wherever possible. However, dynamic projection out of the types *any* and *environment* [13] permits the dynamic binding required for true orthogonal persistence [2]. Napier88 is unusual in that, like its predecessor PS-algol [6], it is a store-based language with higher order procedures [14] and block retention [15]. The Napier88 system consists of the language and its persistent store. This persistent store is populated with objects, some of which are used to support itself. Examples of such tools include an object browser [16], a window manager [17] and the Napier88 compiler, which may be called dynamically to provide ad-hoc polymorphism and reflection.

### **2.2. Persistent Abstract Machine**

The Persistent Abstract Machine (PAM) [18] is a byte-coded interpreter, primarily designed to support the Napier88 programming language. Due to the modularity of its design and implementation, it may be used to support any language with at most the following features: persistence, polymorphism, subtype inheritance, first class procedures, abstract data types, block structure and assignment. Other features, such as object-oriented programming in the Smalltalk style [19] and lazy evaluation, may be modelled at a higher level using the same support mechanisms as first class procedures [14,20]. This covers most algorithmic, object-oriented and applicative programming languages currently in use.

## A Persistent Distributed Architecture Supported by the Mach Operating System

Francis Vaughan, Tracy Schunke, Bett Koch,  
Alan Dearle, Chris Marlin & Chris Barter.

{francis,tracy,bett,al,marlin,chris}@cs.adelaide.edu.au

*Department of Computer Science  
The University of Adelaide  
G.P.O. Box 498, Adelaide  
South Australia 5001  
Australia*

### Abstract

Persistent systems attempt to hide the traditional distinction between short-term and long-term storage from the applications programmer. There are considerable potential benefits if a programmer can operate at a level of abstraction in which this distinction does not exist, and can simply deal with all stored objects as if they come from a single persistent store. An important step towards achieving such an applications programming environment is the provision of a virtual machine layer which provides a so-called persistent architecture. In recent years, some considerable progress has been made towards such persistent architectures, but several of the important remaining research issues concern concurrent access to a persistent store. This paper describes an experimental framework for investigating these issues; the persistent distributed architecture embodied within this framework exploits a number of the facilities provided by the Mach distributed operating system.

### 1. Introduction

In most programming systems, a programmer is forced to manage a multiplicity of resources. Typically, the tasks that the programmer has to perform include the allocation and deallocation of volatile storage for data structures and the mapping of these data structures to long term storage (DBMS or file system). In today's distributed, multiprocessor environments, programmers have an even greater burden placed upon them. In addition to the traditional tasks, they also have to manage processor allocation, network communication and the transmission of data between clients. This paper describes progress on the development of an underlying architecture which relieves a programmer of all these concerns by automatically managing the aforementioned resources.

Persistence is the length of time that data exists and is usable [1]. It has been suggested by others [2,3] that programming systems should support long-lived data objects of arbitrary complexity. Such data objects may not only outlive instantiations of the program that created them, but also outlive versions of the program, or even the useful life of the program in all its versions. Such a programming environment subsumes the roles of file systems and database management systems.

This paper describes an architecture designed to support applications written in the persistent programming language Napier88 [4]. The Napier88 programmer views the world as a graph of strongly typed, stable persistent objects. In this domain of discourse, all the physical properties of data have been abstracted over; examples of such properties include the location of the data (i.e. whether it is on disk or in RAM, and on which machine it resides) and how long the data exists.



`mach_port_request_notification(task, name, which, sync, notify, previous)`

Uses the supplied notify send-once right to request the specified notification for a port right. Returns the previously registered send-once right, if any.

`mach_port_insert_right(task, name, right)` Inserts the supplied send, send-once, or receive right into the target task, giving it the specified name.

`mach_port_extract_right(task, name, how_extracted, OUT right)`

Returns a send, send-once, or receive right extracted from the target task.



### 2.3. Mach

Mach [21] provides a suitable base for our experiments through its support for programmable page fault handling, inter-process communication, exception handling, and multiple threads.

Under Mach, the user is permitted to provide a process called an *external pager* which services page faults. If an external pager is associated with a user process, the Mach kernel will forward page fault exceptions to that external pager, which will return the required data (in the case of a read fault) or may write the data to some stable medium (for pages removed from the client's physical memory). This external pager mechanism implements most of the functionality needed to support the coherent persistent address space described later.

The single inter-process communication (IPC) structure available in Mach permits a transparent interface to be built, independent of the physical location of the communicating parties.

Mach supports more than one thread of execution in a single virtual address space, which is exploited by the architecture described in this paper. We have found this to be especially useful in building asynchronous communication protocols, such as our cache coherency protocol.

## 3. Implementation structure

### 3.1. Overview

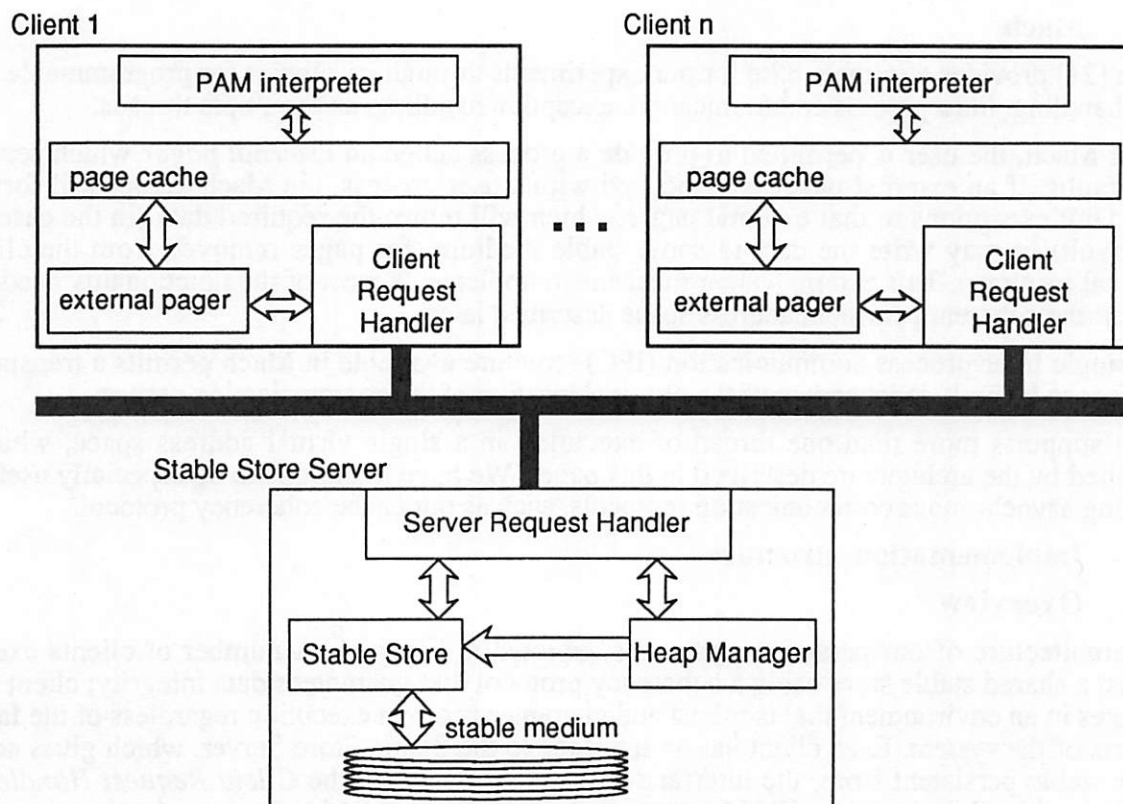
The architecture of our persistent system is depicted in Figure 1. A number of clients execute against a shared stable store using a coherency protocol that guarantees data integrity; client code executes in an environment that is robust and guarantees correct execution regardless of the failure of parts of the system. Each client has an interface to the Stable Store Server, which gives access to the stable persistent store; the interface at the client is called the *Client Request Handler*. In addition, each client contains a PAM interpreter which executes PAM programs (such as compiled Napier88 programs) and a page cache which holds copies of pages required by the interpreter.

Within a client, coherency is maintained by the Client Request Handler and the external pager. Each client maintains copies of pages from the stable store which may be shared and modified by different clients.

The *Stable Store Server* consists of:

- the *Server Request Handler*, which acts as the server's interface to the connected clients,
- the *Stable Store* which manages physical pages,
- the *Heap Manager* which manages the store at a logical level, and
- the *stable medium* representing the physical device.

The Stable Store, which is guaranteed to move from one consistent state to another, contains the complete environment in which clients execute. In addition to the usual passive data found in traditional database and file systems, the Stable Store contains active data including the state of all processes executing within it. This provides the potential for restarting processes found in the persistent store should some element of the system fail. The protocol definition includes the maintenance of structures needed to correctly roll back the execution state of interdependent clients should failure occur in any part of the system. Those parts of the system that can continue without jeopardising the store's integrity are unaffected.



**Figure 1.** A distributed persistent architecture.

Unlike previous Napier88 implementations, in which references to objects were via a persistent identifier (PID) [9,10], object pointers refer directly to the address of objects within the address space. This results in significant simplification of the store architecture, but results in additional complication in the management of a local heap space for each persistent process.

### 3.2. Heap based storage management

One of the most notable features of the Persistent Abstract Machine [22] is that it is constructed entirely upon a heap-based storage architecture. The heap-based architecture was considered advantageous for the following reasons:

- Only one storage mechanism is required, easing implementation.
- There is only one possible way of exhausting the store. In a persistent system, this is an essential requirement, since applications should only run out of store when the persistent store is exhausted, and not merely when one of the storage mechanisms is exhausted.
- The Napier88 language supports first-class procedures with free variables. A direct implication of this is that the locations of variables may persist after their names are out of scope. This property is known as block retention [15] and does not occur in conventional stack-based languages.

Stacks are still used conceptually, with each activation record being implemented as an individual heap object. Each of these records represent the piece of stack required to implement a block or procedure execution of the Napier88 language. The size of each record may be calculated statically, which leads to an efficient use of the available working space.

The persistent store is therefore implemented as one very large heap. All Napier88 objects reside in this heap. A consequence of activation records being allocated from a heap rather than a stack is that the heap is heavily used for allocating store to short lived objects. These short lived objects may be garbage collected provided that transient objects may be distinguished from persistent objects resident in the heap.

Garbage collecting a large object space such as the one in our architecture is potentially a very expensive operation; the implementation therefore uses tactics to make garbage collecting the entire persistent store infrequent. One tactic attempts to prevent short lived objects from entering the persistent heap. This is achieved by allocating, for each process, a local heap of objects which may be garbage collected independently of the whole persistent store. Local heaps may be safely garbage collected provided that no external references (from objects resident in other clients or the server) point into them. Fortunately, the creation and export of such pointers is easily detected making this technique tractable. Note that space for local heaps is allocated from the persistent heap and that pages which make up local heaps are actually part of the persistent store.

### 3.3. Stability and Coherency

Database management systems require a sequence of update operations by a user process to be contained within an atomic transaction. That is, either all modifications are completed, or none are made. Traditionally, such atomicity is achieved by locking portions of the database, so that while a transaction is in progress, no other user process may view modified data [23].

Napier88 provides no language-level synchronization primitives; however, suitable language features have been proposed for the Napier family of languages [24]. The proposal does not force all data accesses to be serialised; instead, anarchic access to the store is permitted. However, the store itself must be kept consistent, which presents two problems:

- the effects of a series of updates during which a failure occurs (either of client or server), must be able to be undone (i.e. *rolled back*), and
- no process must view or act upon out-of-date data, or be able to modify a data item concurrently with another process.

In this system, store stability deals with the first aspect, and the cache coherency protocol with the latter.

## 4. Store Stability

Alternative schemes for sharing virtual memory between distributed clients have been proposed [25,26]. In [27], the underlying data repository moves between *stable states* through *checkpoint* operations. A checkpoint involves flushing those pages onto disk which have been modified since the last checkpoint. A new stable state is achieved when a known set of such pages has been thus secured, and any record of their original contents, as held in the previous stable state, can be safely disposed.

Wu and Fuchs [28] describe a system whereby checkpoints are carried out on individual nodes (i.e. clients), as soon as another node requests the use of any updated data. A major concern of

this work has been to limit rollback propagation, so that the failure of any client only affects that client.

In the architecture described in this paper, information is instead maintained by the central server with regard to the distribution and modification status of pages among all connected clients. No writeback is required until a stabilise operation is instigated by a client.

A client which has modified pages may pass these to another client. Thus, clients may become dependent on one another by virtue of having seen the same data, which has been modified with respect to the Stable Store. We term such dependent clients *associates* and a set of mutually dependent clients an *association*.

It is important to note that associations are dynamic in nature, with clients being added and associations merging over time. These associations are maintained by the Stable Store Server. Linked with each association is a list of page identifiers, which identify those pages modified by members of the association since the last stabilise.

Any client may initiate a stabilise operation; this requires all associates belonging to the initiating client's association to stabilise. All pages on the page identifier list must be returned to the server and written back to the store as an atomic transaction. (This is not necessarily the only time at which modified pages are sent back to the server; the Mach kernel may force the removal of pages from a client's cache at any time.) Several independent checkpoint operations may be in progress at any time since, by definition, an associate can only belong to one association.

Clearly, the failure of one client will affect all members of the corresponding association; these associates are required to return to their previous stable state. In Napier88, the state of a process is stored as a heap object; this information is therefore saved during a stabilisation, and is used to initialise a client should it be required to restart.

To permit rollback, a *shadow paging* technique of writing data to disk is employed [29]. In general terms, this scheme involves maintaining original page versions along with the new. As the final stage in a checkpoint, a single disk-page write updates a mapping, reflecting which version is to be treated as the original henceforth. Until this point, the previous stable state is completely preserved, and indicated by the old mapping. For this reason the store can be described as *self-consistent*.

## 5. Cache Coherency

### 5.1. Communication architectures

The introduction of multiple clients which can concurrently access pages within the Stable Store by operating over a local cache poses a major challenge for the design of a cache coherency protocol. The bus-based multiprocessor hardware cache coherency methods (e.g. write-through, copy-back, etc.) [30] were considered, but rejected as too inefficient. For example, the client may be operating over a distributed network, where the communication costs are more expensive than in a bus-based multiprocessor, and so a different scheme is needed.

The coherency protocol has been specified as two finite state automata, representing the states of a page within each of the clients and the server. Client-server interaction is modelled by transition interactions between the two automata. Client-client interaction is represented within the client's automaton. In the implementation, finite state machines based on these models are used to maintain page coherency in the entire system.



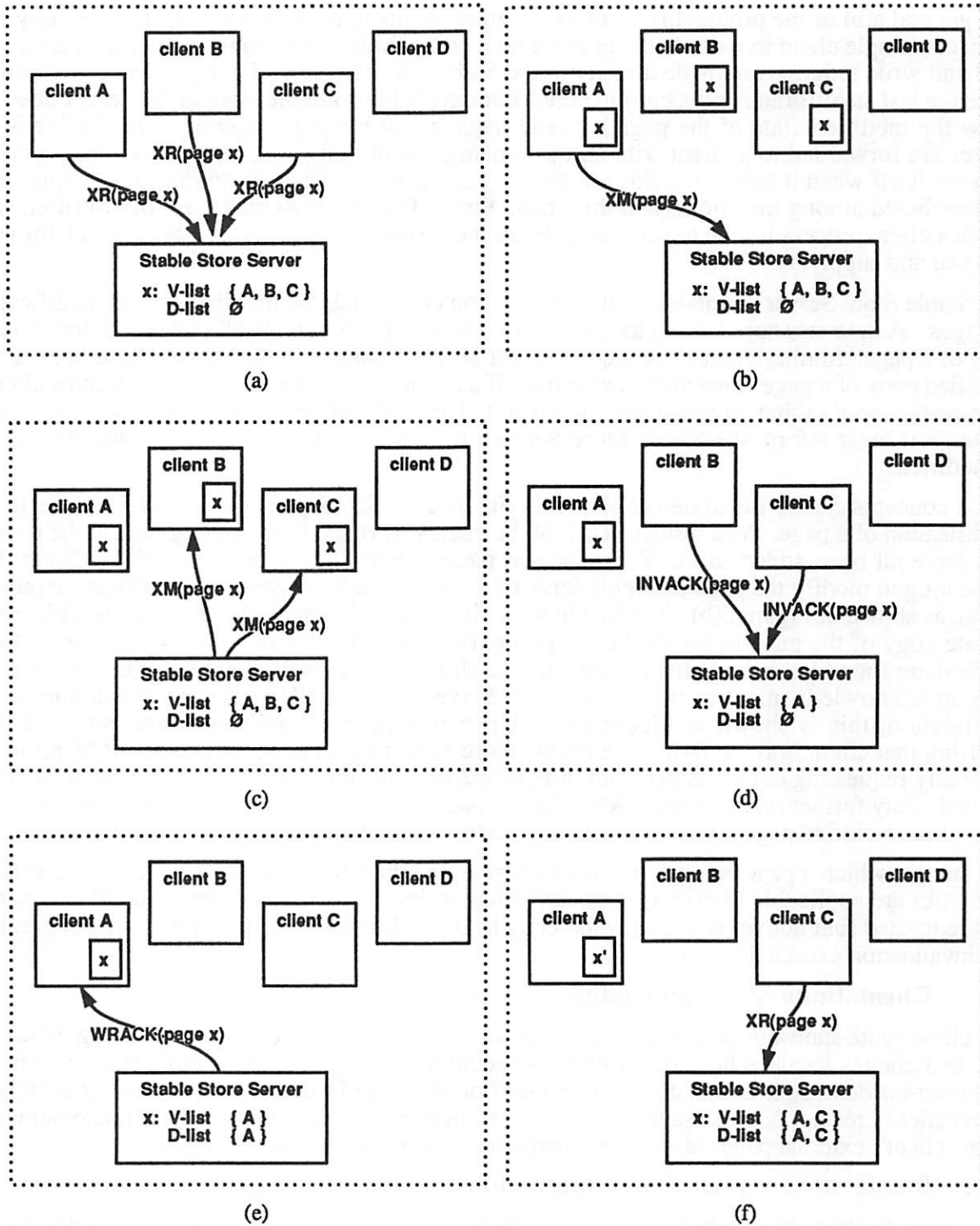


Figure 2. The modification protocol for a shared page.



The general aim of the protocol is to allow multiple clients to read the most up-to-date copy of a page, or a single client to write to the page without compromising the coherency of the pages. All read and write requests are made directly to the Stable Store Server. If a page has been modified since the last stabilisation and a current copy is not available from the store or the server does not know the modified state of the page, all read requests for the page arriving at the Stable Store Server are forwarded to a client with an up-to-date copy of that page. The server only services requests itself when it knows it holds a valid copy of the page. Thus, up-to-date page copies may be distributed among the clients and the Stable Store. The aim is to maximise the freedom with which a client process is able to run, and prevent the server from becoming a bottle-neck for page retrieval and supply.

The Stable Store Server maintains all the information concerning the distribution and modification of pages. A data structure known as the *V-list* contains the identity of all clients holding a valid copy of a page. Similarly, the dependency list, or *D-list*, records which clients hold or have held a modified copy of a page since the last stabilise. If a client wishes to modify a page, it must already have read access to that page. If the page is not shared, the client may freely modify the page, although it must inform the Stable Store Server of the modification if the page was previously unmodified.

These concepts can be illustrated as shown in Figure 2, which depicts the events involved in the modification of a page. As a result of each of the clients A, B and C attempting to read the page x, they have all been added to the V-list for that page; this is shown in Figure 2(a). Client A is attempting to modify the page and must forward to the Stable Store Server a modification request (XM), as shown in Figure 2(b). The Stable Store Server next instructs all other clients with an up-to-date copy of the page to invalidate their copy (via an XM); the identities of the clients to be notified are found from the V-list for the page, as shown in Figure 2(c). These clients must reply with an acknowledgement to the Stable Store Server (via an INVACK) on completion of the invalidation; this is shown in Figure 2(d). Upon receipt of all acknowledgements and after inserting that client into the D-list, the Stable Store Server sends a signal (via a WRACK) to the originally requesting client, as depicted in Figure 2(e), indicating that write permission has been granted. Any further read requests (XRs) for the page result in the requesting client receiving the up-to-date, modified page copy and thus being inserted into the V- and D-lists for that page.

The state to which a page belongs in both the server and the clients indicates which of the various properties are applicable to the page at that particular time. The states include: modified, shared, read requested (but not yet resident), valid copy held, modification requested (but not yet granted) and invalidation expected.

## **5.2. Client finite state automaton**

The client finite state automaton represents the set of states to which a page may belong while it is held in a client's local cache. Any page not resident in a client's cache is considered to be in the start (non-resident) state in that client. The transition of a page from one state to another is initiated by the client's receipt of a message for that page from either the Stable Store Server, another client or this client's external pager (due to the interpreter's attempt to access the page).

## **5.3. Stable store server finite state automaton**

The server finite state automaton represents the various states to which a page, exported by the Stable Store Server, may belong. Any page which has not been exported from the Stable Store

Server implicitly belongs to the start state. Transitions are initiated by requests from the connected clients.

Associations are constructed between stabilise cycles in the server. If the server receives a read request, the requesting client is added to the page's V-list. If the page is modified, the client is also inserted in the page's D-list and the client and its current associates must be merged with the association to which all other members of the page's D-list belong. This is necessary because the requesting client is dependent upon the modified page copy which the other D-list members have accessed.

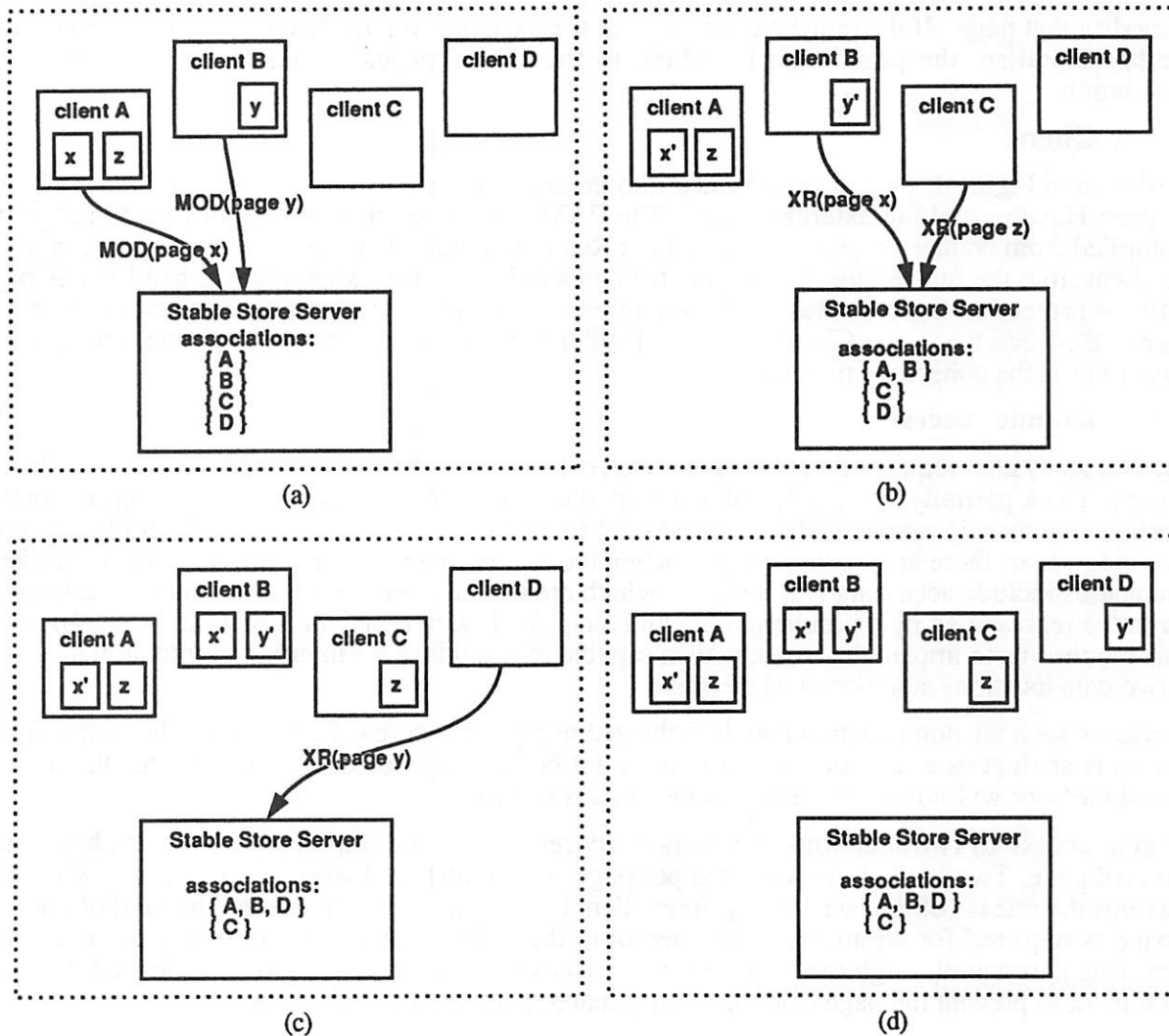


Figure 3. Expanding associations.

Figure 3 illustrates a combination of requests which lead to the expansion of an association. Firstly, Figure 3(a) depicts a collection of clients which do not share any pages; thus each association consists of precisely one client. Figure 3(a) also shows that client A modifies  $x$  and client B modifies  $y$ . Figure 3(b) depicts clients B and C each requesting pages held by client A;

since client B is requesting page x, which has been modified by client A, the Stable Store Server merges the associations containing clients A and B. The association containing client C is not merged with this association, as the page requested (page z) is not modified. Thus, the failure or stabilisation of client A or C will not affect the other. A further request from client D for another modified page (page y) is shown in Figure 3(c); since this page has been modified by client B, it causes the association containing client D to merge with that containing A and B. The result of this interaction is shown in Figure 3(d).

The server may receive a modification signal indicating either the only client holding a copy of a page is modifying that page for the first time, or one of a number of clients sharing a page wishes to modify that page. If the request indicates that the page is being modified for the first time since the last stabilise, the page must be added to the list of pages linked to the corresponding association.

## **6. Client**

As shown in Figure 1, the client is divided into three main threads: the PAM interpreter, the Client Request Handler and the external pager. The PAM interpreter simply executes PAM programs (compiled from Napier programs). The Client Request Handler handles all incoming messages to the client from the Stable Store Server and from other clients. The external pager handles any page faults or protection faults caused by the interpreter's attempts to access non-resident or protected pages. Between them, the Client Request Handler and the external pager implement the client's part of the cache coherency protocol.

### **6.1. Atomic access**

The Napier system requires that updates to objects be atomic. This is necessary so that an object is never left in a partially modified, and hence inconsistent, state. The interpreter acts upon objects mostly in an atomic manner. Most accesses are to 32-bit words, which is atomic at the machine level. However, there are some occasions when the atomicity provided at this level is insufficient. Such cases include accessing real numbers which are 64-bit quantities and referencing variants. A variant is represented by a pointer to data and a tag field, which may be stored non-contiguously. This requires us to implement a mechanism capable of providing the interpreter with atomic access to two data locations at arbitrary addresses.

Providing such an atomic action may be achieved using a structure called a *latch*. The semantics of a latch is analogous to a door latch: it may be set before the door is closed, but once the door is closed the door will not open again until the latch is released.

Atomic access to two locations at arbitrary addresses may be implemented via latching each affected page. Two latches are provided per page – read latch and write latch. A latch, when set, prevents the release of the page to any other client for the purpose indicated by the kind of latch. If a page is required for an atomic read operation, the write latch is set and so a write operation occurring part way through the read operation is prevented. If an atomic write is desired, the read latch is set to prevent the page from becoming shared part way through the write.

The design of the latching mechanism has aimed for efficiency, particularly for trivial cases, such as when only one page is needed and the page is already resident. When the need arises to access more than one page, pages are latched serially and in a defined order (in fact, ascending address order) to prevent circular dependencies with competing clients, and hence avoid deadlock. Once all of the necessary pages are in the appropriate state and with the appropriate latches set, the atomic access is performed and the latches released.

The need for costly mutex locks is obviated, since latches are:

- only ever set and released by the interpreter,
- released by the Client Request Handler when the interpreter is guaranteed to be blocked, and
- read by the Client Request Handler.

## **6.2. External Pager**

The external pager handles all page accesses by the interpreter which result in protection exceptions or page faults. If the interpreter attempts to access a protected or non-resident page, an invalid access exception or page fault will be raised. The kernel fields the access attempt and generates a message that is delivered to the external pager via the external pager's port. From this, it is trivial to determine the appropriate page and the state in which the interpreter expects to find the page upon return. The external pager must cause the appropriate state transition to permit the required access. This may involve a simple state change or dialogue via IPC calls between the Client Request Handler and the Stable Store Server. Upon completion, the external pager replies to the kernel which, in turn, reschedules the interpreter. The interpreter will retry and successfully execute the instruction which originally caused the exception or page fault.

## **6.3. Page removal by the kernel**

The external pager also handles the return of modified pages to the Stable Store (to relieve pressure on local physical memory). If a removed page is modified, an up-to-date copy must be returned to the Stable Store. If a removed page is not modified, the Stable Store Server must be notified that this client no longer holds a valid page copy.

Normally, the Mach kernel only informs the external pager of the removal of a modified page. In order to receive information on the removal of all pages (modified and unmodified), the external pager must ensure that all pages are modified (non-destructively) when they are brought in to the client. This is implemented by initially protecting such a page from all accesses. The interpreter then resumes execution, causing a second exception, at which time the page is unprotected and non-destructively modified. In order to avoid deadlock, it is necessary to wait until the second exception before attempting the modification, as the external pager cannot attempt to modify a page before ensuring the kernel has linked the page into its internal data structures.

## **6.4. Interpreter**

Ideally, the interpreter should not be aware of the existence of the other parts of the client, only perceiving a single, flat, virtual address space. In reality, a few concessions must be made. The whole address range cannot be made available to the persistent heap since a small area is required within which to place both the interpreter and the coherence mechanisms. This will be reasonably small compared to the entire address space. This area is demand-paged by the default pager since it is not persistent.

## **6.5. Client Request Handler**

The Client Request Handler is the client's interface to the outside world. Some requests generated within a client are also passed to the Client Request Handler's port. The cache coherency protocol requires that messages between particular pairs of communicants are delivered in the order in which they are generated. To maintain temporal ordering, all external communications relating to a resident page must be passed through the Client Request Handler. The Client Request Handler is



also responsible for maintaining the appropriate state information caused by external events via the coherency protocol.

The responsibility for startup and creation of the other threads in the client rests with the Client Request Handler. It must also handle client termination and restart.

### 6.6. Local Heap Management

Each client maintains a local cache of pages. The pages within this cache are simply up-to-date page copies, or modifications of page copies, which have been received from the Stable Store Server or from another client. Each PAM process executing in a client maintains a *local heap* for local object creation; this is a previously unused set of (contiguous) persistent pages. Local heaps may be independently garbage collected; to allow such garbage collection, all references into a local heap must be contained within the client. Any page containing references into a local heap must not be exported from the client until the referenced objects have been removed from all local heaps. A method based on generation based garbage collection [31] is used to overcome this problem [32].

## 7. Stable Store Server

All access to the persistent store is controlled by the entity known in its entirety as the Stable Store Server. It has two main functions: it supplies pages upon demand to clients, ensuring that coherent versions of the pages are supplied, and it is responsible for maintaining the integrity of the Stable Store. It also allocates ranges of the persistent address space to processes and garbage collects the main heap.

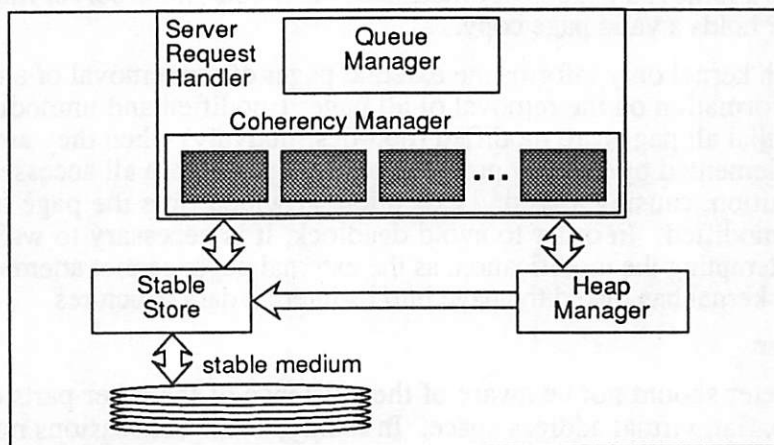


Figure 4. The Stable Store Server.

Communications from clients are received by the Server Request Handler as shown in Figure 1 above. Figure 4 depicts the Stable Store Server in more detail. The Server Request Handler is composed of:

- the *queue manager*, which accepts incoming Mach IPC messages, maintains internal message queues and distributes messages to other components of the Stable Store Server, and
- the *coherency manager*, which implements the server side of the cache coherency protocol.



The *Stable Store Manager* is invoked whenever existing pages are read from or written to the store. A request for fresh space, on the other hand, is directed to the *Stable Heap Manager*. A checkpoint operation demands special co-operation from all components. Ultimately, the Coherency Manager is responsible for co-ordinating the necessary activities and dispatching replies to the clients.

### 7.1. Queue Manager

This thread has the simple task of removing IPC messages from the input port of the Stable Store Server, extracting the message content and adding it to an internally managed queue. The messages are passed to the coherency manager upon demand.

A queue manager is required for several reasons. Firstly, if the queue associated with a Mach IPC input port becomes full, any processes sending messages to that port will block; providing an internal queue reduces the likelihood of clients blocking.

An internal queue also enables the selective servicing of signals. In general, messages are dealt with in arrival order, but while a stabilise is in progress, it is expedient and sometimes imperative that requests are delayed from clients not stabilising. At such times, the coherency manager may request the queue manager to delay a message until a more suitable time.

A further benefit of an internal queue is the isolation of the Mach IPC system from the server code. Finally, the provision of an internal queue anticipates increasing concurrency within the Stable Store Server.

### 7.2. The Coherency manager

The actions taken by the Coherency Manager to service a request must both adhere to the dictates of the corresponding finite state machine, and maintain the information required for cache coherency control. To implement the coherency protocol, the Coherency Manager thread manipulates several data structures.

Among these data structures is the *Export Table*, which contains an entry for every page exported to clients by the Stable Store and is organised as a hash table. Each entry holds a page identifier which is the hash key, a state, and the V- and D-lists mentioned earlier. These latter lists can now be explained in more detail:

- The V-list records which clients currently hold a valid copy of the page. This information is used to determine which clients can forward an up-to-date copy of a page to a requesting client. It is also used to determine which clients must be sent invalidation signals when one of the clients in the V-list requests modification permission.
- The D-list records those clients which have seen a copy of the page modified with respect to the Stable Store. This is used in the maintenance of associations.

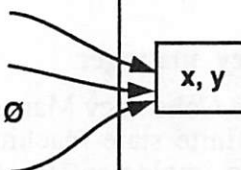
The state field refers directly to the finite state machine: for each valid combination of signal and state, a specified sequence of actions accompanies a possible transition to a new state. For example, the entries shown in Figure 5(a) below correspond to the state of the pages in Figure 3(d).

Internally, clients are designated by a unique number, representing a single bit position; this allows sets of clients to be efficiently represented. This number is also used to index the *Connected Clients Table*, which contains the address of the client's input port. In addition, each entry

contains a bit vector representing the client's association and a list of all modified pages held by any of the associates since the last stabilisation; this list is known as the association's *check list*. The Connected Clients Table is illustrated in Figure 5(b), which shows the information recorded in the case of Figure 3(d).

pageId	state	V-list	D-list
x	SH_MOD	1100	1100
y	SH_MOD	0101	0101
z	SH_NONMOD	1010	0000

(a)

portId	association	check list pointer
portA	1101	
portB	1101	
portC	0010	
portD	1101	

(b)

**Figure 5.** The Export Table and Connected Clients Table.

The Stable Store Server includes another locally managed queue, called the *XR queue* (external read queue). External read requests for pages for which write permission is currently being negotiated, are diverted to this queue, and forwarded to the writer after write acknowledgement has been granted.

### 7.3. Stable Store

This module is responsible for the transfer of data to and from the stable medium. It implements the shadow paging mechanism, described in Section 4, allowing the system to recover from soft failures. In the event of a roll back by an association, only those pages contained in that association's check list must have their shadowing records removed.

### 7.4. Heap Manager

The store module manages the stable virtual address space at a physical level. On the other hand, the Heap Manager is responsible for the logical management of this space. Tasks performed by the Heap Manager therefore include the allocation of space, its deallocation and garbage collection.

When a client requests free space, the Heap Manager responds by allocating unused pages. Object placement within these pages is performed by clients.

Newly allocated pages are classed as modified, since the first operation on such a page will be a write. Consequently, they will be placed on appropriate page check lists in the Connected Clients Table as described in Section 7.2. Conversely, these pages are extracted from the relevant page check lists when an association rolls back.

From the above discussion, it would appear that the Stable Store Server understands nothing of objects or the contents of the pages it is required to supply and secure. This is not quite true, on two accounts.

- All internal information regarding the state of the heap of pages must be persistent. As part of a checkpoint operation, the Heap Manager saves its control variables by writing to the *heap root object*, which is stored at a distinguished position in the stable virtual address space.
- Garbage collection must, by necessity, be carried out at the object level.

## 8. Conclusion and future plans

An experimental distributed persistent architecture has been described. This architecture consists of a central server supplying pages from a persistent store to a collection of clients. Each client has a cache of pages on which its interpreter may operate and pages held within clients or the central server may be invalidated by changes elsewhere in the distributed system. A sophisticated cache coherence protocol ensures that access to a particular object anywhere within the system always retrieves the most up to date copy of it. Various facilities provided by the Mach operating system have been used during this work, including the IPC mechanism and the ability to provide an external pager to service page faults.

Our work with Mach has highlighted some deficiencies in the current form of the operating system. The kernel must, as part of its memory management duties, occasionally remove pages from a user's memory cache. If the page has been modified, the kernel will return the page to the external pager. However, in the current version of Mach if a page is unmodified, the kernel assumes that the page may be removed, without informing the external pager. In this system, although the page may be unmodified with respect to the client's kernel, the page may have been modified with respect to the Stable Store by a different client. This means that the system could potentially lose the only copy of a modified page.

It has been proposed to add extra functionality to the external pager interface to inform the Mach kernel that a page is "precious" (in Mach parlance) and must not be removed without informing the external pager. Currently this is unavailable. Instead, as described in Section 6.3, the coherency manager non-destructively modifies any page supplied to a client, thus forcing the kernel to supply the page back to the external pager when it removes it from the cache.

Another deficiency is that the exception handler messages do not include the type of access that caused the exception. That is, if an area of virtual memory is made inaccessible by use of the `vm_protect` call, the exception message resulting from the attempted access does not indicate whether the access was a read or a write. This does not affect our current system, but represented a potential difficulty in other designs considered.

A further inconvenience is the kernel's removal of pages according to its own LRU algorithm. It would be more useful if the kernel informed the external pager to remove one or more pages,

rather than sending its own choice of pages on to the external pager for removal. This is due to the fact that the pages selected may contain pointers into the client's local heap area, in which case removal is a costly operation. The external pager can determine more appropriate candidates for efficient page removal through the available state information.

The current design has a single interpreter executing against each local cache. To allow multiple instantiations of the interpreter to execute against a single page cache, the atomic access protocol must be modified to work within a single client as well as between clients. This is achieved by adding mutual exclusion locks to the latches used for atomic access.

The notion of many persistent processes distributed across a number of clients makes failure recovery much more difficult. Further work is needed in analysing the semantics of process/client binding and sensible recovery algorithms for them.

Currently, the main heap is managed as a semi-space, to allow the use of a simple compacting garbage collector. This results in a halving of the space available for object storage. It is possible to build a garbage collection system that compacts from one store to another, allowing one store to use the entire address space. This garbage collector would execute against two separate address spaces (each of 4GB).

As noted above, the current design is built around the notion of a single central server. It is clear that it is desirable to allow the Stable Store to be distributed. However, this is greatly complicated by the need to constrain the failure semantics of the system, and difficulties with garbage collection.

The design of the architecture described in this paper is now complete and coding has begun.

### Acknowledgements

This work was partly supported by ARC grant number 4900-6830-1000. We would like to thank the Persistence Project at the University of St Andrews for their continuing cooperation in this work.

### References

- [1] Atkinson M.P. "Programming Languages and Databases". Proceedings 4th VLDB, pp. 408-419 (1978).
- [2] Atkinson M.P., Bailey P.J., Chisholm K.J., Cockshott W.P. & Morrison R. "An Approach to Persistent Programming". The Computer Journal, 26,4 pp. 360-365 (1983).
- [3] Baltzer R. "Living in the next generation operating systems". Proceedings 10th IFIP World Congress, Dublin, Ireland, pp. 283-291. North-Holland, Amsterdam (September 1986).
- [4] Morrison R., Brown A.L., Connor R. & Dearle A. "The Napier88 Reference Manual". Universities of Glasgow & St Andrews Persistent Programming Research Report 77-89 (1989).
- [5] Brown A.L. "Persistent Object Stores". Ph.D Thesis, University of St Andrews, St Andrews, Scotland (1988).
- [6] "The PS-algol Reference Manual fifth edition". Universities of Glasgow & St Andrews Persistent Programming Research Report 12-88 (1988).



- [7] Albano A., Cardelli L. & Orsini R. "Galileo: A Strongly Typed, Interactive Conceptual Language". *ACM Transactions on Database Systems*, 10,2 pp. 230-260 (1985).
- [8] Atkinson M.P., Chisholm K.J. & Cockshott W.P. "CMS – A Chunk management system". University of Edinburgh Internal Report CSR-110-82 (April 1982).
- [9] Cockshott W.P., Atkinson M.P., Chisholm K.J., Bailey P.J. & Morrison R. "POMS: a persistent object management system". *Software Practice and Experience*, 14,1 (January 1984).
- [10] Brown A.L. & Cockshott W.P. "The CPOMS Persistent Object Management System". Universities of Glasgow & St Andrews Persistent Programming Research Report 13-85 (1985).
- [11] Atkinson M.P., Morrison R. & Pratten G. "Designing a persistent information space architecture". *Proceedings 10th IFIP World Congress*, Dublin, Ireland, pp. 115-120 North-Holland, Amsterdam (September 1986).
- [12] Cardelli L. & Wegner P. "On understanding types, data abstraction and polymorphism". *ACM Computing Surveys*, 17,4 pp. 471-523 (December 1985).
- [13] Dearle A. "Environments : A flexible binding mechanism to support system evolution". *Proceedings 22nd Hawaii International Conference on System Sciences*, Hawaii, U.S.A., pp. 846-855 (Jan 1989).
- [14] Atkinson M.P. & Morrison R. "Procedures as Persistent Data Objects". *ACM TOPLAS*, 7,4 pp. 539-559 (October 1985).
- [15] Berry D.M. "Block Structure: Retention or Deletion?". *Conference Record of the Third Annual ACM Symposium on the Theory of Computing*, Shakes Heights, Ohio, pp. 86-100 (May 1971).
- [16] Kirby G. & Dearle A. "An adaptive browser for Napier88". (1990). University of St Andrews Research Report CS/90/16 (1990).
- [17] Cutts Q., Dearle A., Kirby G. & Marlin C. "WIN: A persistent window management system". Universities of Glasgow & St Andrews Persistent Programming Research Report 73-89 (1989).
- [18] Morrison R., Brown A.L., Carrick R., Connor R. & Dearle A. "The Persistent Abstract Machine". Universities of Glasgow & St Andrews Persistent Programming Research Report 59-88 (1988).
- [19] Goldberg A. & Robson D. "Smalltalk-80. The Language and its Implementation" Addison-Wesley (1983).
- [20] McNally D.J., Davie A.J.T. & Dearle A. "A Scheme for compiling lazy functional languages". *Proceedings 2nd International Workshop on Implementation of functional programming*. Aspenås, Sweden (September 1988).
- [21] Acceta M., Baron R., Bolosky W., Golub D., Rashid R., Tevanian A. & Young M. "Mach: A New Kernel Foundation for UNIX Development". *USENIX*, pp. 93-112 (July 1986).



- [22] Connor R., Brown A.L., Carrick R., Dearle A. & Morrison R. "The Persistent Abstract Machine". Proceedings 3rd International Workshop on Persistent Object Systems, Newcastle, New South Wales, pp. 80-96 (January 1989).
- [23] Eswaran K.P., Gray J.N., Lorie R.A. & Traiger I.L. "The notions of consistency and predicate locks in database systems". Communications of the ACM, 19,11 pp. 624-633 (November 1976).
- [24] Morrison R., Barter C., Brown A.L., Connor R., Carrick R., Dearle A., Hurst J. & Livesey M. "Language Design Issues in Integrating Process Oriented Computation with a Persistent Environment". Proc 22nd Hawaii International Conference on Systems Sciences, Hawaii, pp. 736-744 (January 1989).
- [25] Li K. & Hudak P. "Memory Coherence in Shared Virtual Memory Systems". ACM Transactions on Computer Systems, 17,4 pp. 321-359 (November 1989).
- [26] Rosenberg J., Henskens F., Brown F., Morrison R. & Munro D. "Stability in a Persistent Store Based on a Large Virtual Memory". University of St Andrews Research Report CS/90/6 (1990).
- [27] Morrison R., Brown A.L., Connor R. & Dearle A. "Napier88 Release1.1". St Andrews University, St Andrews, Scotland (1989).
- [28] Wu K.L. & Fuchs W.K. "Recoverable Distributed Shared Virtual Memory". IEEE Transactions on Computers, 39,4 pp. 460-469 (April 1990).
- [29] Lorie A.L. "Physical Integrity in a Large Segmented Database". ACM Transactions on Database Systems, 2,1 pp. 91-104. (1977).
- [30] Archibald J. & Baer J.L. "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model". ACM Transactions on Computer Systems, 4, pp. 273-298 (November 1986).
- [31] Ungar D. "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm". ACM SIGPLAN Notices, 9,5 pp. 157-167 (May 1984).
- [32] Koch B., Schunke T., Dearle A., Vaughan F., Marlin C., Fazakerley R. & Barter C. "Cache Coherency and Storage Management in a Distributed Persistent System". Proceedings 4th International Workshop on Persistent Object Systems, Martha's Vineyard, *to appear* (September 1990).

# A Trusted X Window System Server for Trusted Mach

Jeremy Epstein  
(epstein@trwacs.fp.trw.com)

Marvin Shugerman  
(shugerman@trwacs.fp.trw.com)

TRW Systems Division  
1 Federal Systems Park Drive  
Fairfax, Virginia 22033-4417  
703-876-8776

August 30, 1990

## Abstract

The MIT X Window System<sup>1</sup> is an industry-standard windowing system which is in wide use. We have performed a preliminary security analysis of the X Window System, and determined that it is feasible to develop at the B3-level<sup>2</sup> a Trusted X (TX) system while maintaining a high degree of compatibility with the existing X Window System.

We have chosen as our base operating system the Trusted Mach (TMach) kernel which is a prototype of a B3-level Mach system.

This paper describes our prototype of a TX Server on Trusted Mach. The architecture of the server, its interactions with other TMach servers and the trusted window manager, and its use of TMach services are all described. In addition, conversion of the MIT-developed, UNIX<sup>3</sup> system based X Window System sample server to TMach is described. Our prototype is a work-in-progress, designed to evolve to a B3 system.<sup>4</sup>

<sup>1</sup>X Window System is a trademark of the Massachusetts Institute of Technology.

<sup>2</sup>B3 is one of the highest levels of security defined by the Trusted Computer System Evaluation Criteria (TCSEC). From least to most secure, the TCSEC levels are D, C1, C2, B1, B2, B3, A1.

<sup>3</sup>UNIX is a registered trademark of AT&T.

<sup>4</sup>This project is sponsored by the Defense Advanced Research Projects Agency under Contract No. MDA 972-89-C0029.

# 1 Introduction

The MIT X Window System is an industry-standard windowing system which is in wide use. We have performed a preliminary security analysis of the X Window System, and determined that it is feasible to develop at the B3-level<sup>5</sup> a Trusted X system while maintaining a high degree of compatibility with the existing X Window System.

The key to maintaining compatibility is the architecture of the X Window System itself, which relies upon a client-server architecture. Specifically, one or more application programs — called clients — communicate with a server which may reside on the same host as the applications or be distributed across a network. Our analysis of the X protocol has indicated that only a few extensions need be added to support secure operation; the majority of the engineering effort is relegated to a Trusted X Window System Server (TX Server) with an associated trusted window manager and TX Trusted Shell.

We have chosen as our base operating system, the Trusted Mach (TMach) kernel [3] which is a prototype of a B3-level Mach [2] system. TMach is currently being developed jointly by Trusted Information Systems and the Open Software Foundation (OSF) and will be the basis for the OSF's second product offering, OSF/2. These three projects (Mach at CMU, TMach at TIS, and our Trusted X Window System work) are all being supported by DARPA as part of their advanced trusted systems program.

The TX Server must be a TMach trusted application since each client may be executing at a different security level. Each client communicates with the TX Server over a port pair; the security level of the port pair can be changed dynamically via X protocol extensions.<sup>6</sup> As such, the TX Server must enforce TMach's mandatory access control policy.

The design of the TX Server is further complicated due to the number of interfaces it requires. During normal operation, the TX Server interacts with the TMach File Server to access fonts stored in the file store. At any time, the user may depress the secure attention key sequence to invoke the TX Trusted Shell process which itself interacts with other trusted TMach servers. Finally, should auditable events be generated during execution, the TX Server sends the audit record(s) to the TMach Audit Server for subsequent logging.

The X Window System provides graphical mechanisms rather than policies. Particular graphical styles can be generated by clients or by window managers which

<sup>5</sup>For a definition of B3 and other security related terms, see [1].

<sup>6</sup>More accurately, the protocol extensions change the level of the data coming through the port. The security level of the port is constant.

decorate windows drawn by other clients. Any window system which complies with the X protocol conventions can be used with an untrusted X server. However, only trusted window managers may be used with a TX Server since they must manipulate windows at differing security levels.

This paper describes our work-in-progress towards building a prototype of a TX Server on Trusted Mach. We start with an overview of X and TMach concepts and then describe the architecture of the MIT sample server. Next, we indicate necessary changes in TMach to support TX and the architecture of the TX server itself. Only those architectural changes which impact TMach are discussed. We conclude with a discussion of related work.

## 2 X Concepts

In this section we discuss the concepts used in the X Window System as distributed by MIT and various computer vendors.

The X Window System is based on the notion of distributed computing. As shown in Figure 1, the *X server* manages the screen(s), keyboard, and pointing device (typically a mouse).

*X clients* and the X server communicate via the X protocol [4]. Clients send *requests* to the server over a bi-directional reliable communications channel (using, for example, TCP/IP or DECnet), and receive *events* and *responses*. Errors are a particular kind of response. Protocol requests are typically asynchronous, since most of them have no reply. Protocol requests include administrative requests, requests to create and destroy resources (defined below), and drawing requests.

The X server manages *X resources* on behalf of the clients. Resources include:

- windows, which are the basic drawing unit
- pixmaps, which are graphical images which exist only in memory
- fonts, which are a read-only matrix of glyphs, typically characters
- cursors, which are the visible shape of the pointer on the screen
- graphics contexts, which define the characteristics used for drawing, such as line styles, fill types and patterns, and fonts
- atoms, which are unique identifiers
- properties, which hold arbitrary data by specifying a window and an atom.

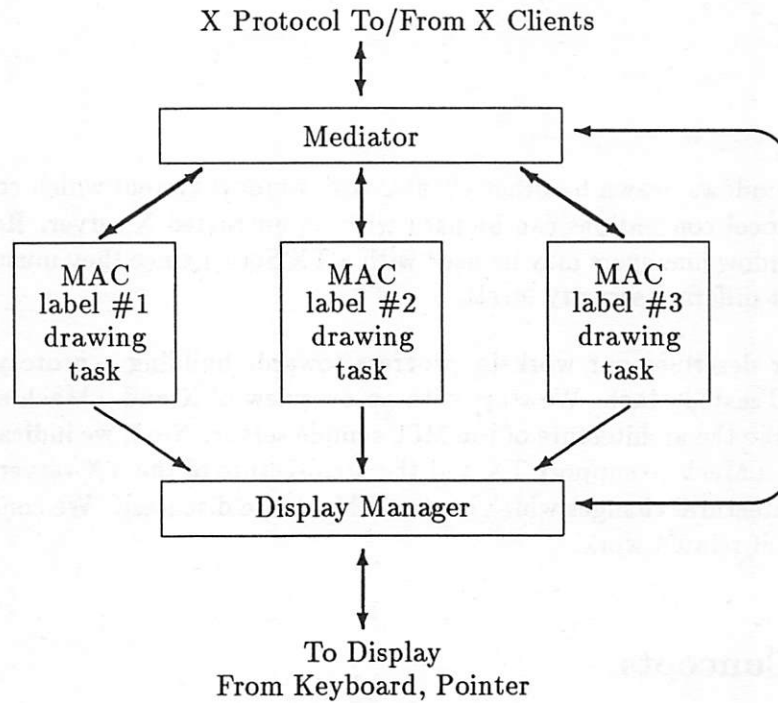


Figure 1: X Architecture

There are also several global resources (such as the search path for fonts and the keyboard and pointer characteristics) which may be changed, but are never created or destroyed. Resources are referred to by resource IDs which are associated by clients (and not the server) with newly created resources.

The X server can be compared to a file system, where resources are files. In a traditional file system, files are opened and subsequent operations use a file handle or descriptor. This allows access control to be checked only when a file is opened. However, in X, each protocol request refers to required resources via their resource IDs. As will be seen later, this means that access control must be enforced on every protocol operation.

X clients can generate protocol requests directly. However, the Xlib library (described in [5]) provides a slightly higher level view of the protocol, including a subroutine interface (which provides generation of the required byte stream for each protocol request) and some abstractions. More commonly applications are written using a toolkit such as Xt (described in [6]) and a widget set such as Athena [7] or OSF/Motif<sup>7</sup> [8]. All of the libraries and widget sets are simply abstractions built on top of the protocol. Consequently, their use is invisible to the server.

X has no concept of privilege, and a minimal notion of protection. Protection is provided at connection time only. The X server maintains a host access list which identifies those computers from which connections will be accepted. In addition, an (optional) authentication mechanism allows the server to demand some form of authentication from the client (e.g., an MIT magic cookie or a Kerberos [9] authentication ticket). Once a client is connected to the server, it may perform any request, including a request to turn off authentication for additional clients. Clients can

<sup>7</sup>OSF/Motif is a trademark of the Open Software Foundation.



also directly impact other clients (e.g., by killing them), although such behavior is considered undesirable (see [10]).

Management of windows on the screen is performed by a *window manager*. There are many existing window managers, each of which provides a different look-and-feel. Because there is no notion of privilege in X, the window manager is simply another client. The conventions described in [10] are used to define an environment where “well-behaved” clients can interact cooperatively.

### 3 TMach Overview

The TMach operating system [3] is a prototype trusted version of the Mach operating system. TMach is targeted at the TCSEC [14] B3 level of trust.

The current TMach prototype is based upon kernel modifications made to Mach 2.5. A collection of trusted servers and tasks which provide the required security features resides upon this base. This layering is necessary to meet B3 minimality and modularity requirements. These trusted components include:

- Name Server: the overall server for named objects<sup>8</sup>
- Audit Server: records audit data, manages the audit logs
- Authentication Server: used for maintaining user data such as the user ID number for a user, permissible security levels, etc.
- File Server: the file system
- TMach Trusted Shell: the trusted path of communication between users and the trusted computing base

Additional servers are planned, but are not relevant to this discussion.

### 4 X Sample Server Architecture

The MIT sample server (henceforth “the server”) runs as a single UNIX process. It receives protocol requests from clients and processes them, sending back responses and events. In addition, the server manages keyboard and pointer input. Thus, the server performs its own scheduling to alternate among its activities.

---

<sup>8</sup>Since they have a publically-known name, named objects can be manipulated by subjects at different levels; thus, access to them must be controlled.

The server consists of approximately 80,000 lines of C code<sup>9</sup>, divided into two main sections: Device Independent X (DIX) and Device Dependent X (DDX) [11], [12]. In addition, there is a small operating system dependent section, known as OS (about 7,000 lines of C for the UNIX version).

DIX is machine and display independent, and consists of about 23,000 lines of C. It interprets the protocol, manages the resources, etc.

DDX is machine and display dependent. Its size depends on the particular hardware being used. For the Sun 3, it is about 53,000 lines of C. It performs all of the physical drawing, manages the keyboard and pointer, etc.

There is a high degree of interaction between DIX, DDX, and OS. While conceptually DIX sits on top of DDX and OS, the three are not strict "layers"; there are many occasions where DDX and OS call each other and DIX.

## 5 Interaction with TMach

Building a Trusted X system on TMach requires a large number of changes to X, and some changes to TMach. This section describes the necessary changes and provides a summary of the interfaces between TX and TMach. The areas addressed are authentication, mandatory access control (MAC), discretionary access control (DAC), privilege, audit, and font management. In addition we discuss areas where the current TMach prototype is missing facilities needed for the Trusted X system.

### 5.1 Authentication

There are two types of authentication of interest in X: user authentication (i.e., login) and client authentication.

X provides a protocol for logging a user into a system (see [13]). Integration of this protocol with TMach would require interaction with the Authentication Server. Examination of the specific changes required is beyond the scope of the TX project.

A trusted X system needs some method for clients to be authenticated to servers. This allows the user to control access to the display. For the TX project, we rely on TMach to provide the necessary authentication. That is, we expect that the TMach Name Server will (via MAC and DAC) restrict establishment of connections by clients to a TX server to those allowed by the TMach mechanisms.

---

<sup>9</sup>Including comments, but not including header files.

## 5.2 Mandatory Access Control

Multi-level secure systems (those at TCSEC B1 and above) require a mandatory access control policy on subjects and objects. The MAC value associated with a subject or object is called the *MAC label*. A MAC label can be thought of as the sensitivity of the data (e.g., unclassified, secret, top secret), and a MAC policy is a set of rules governing information flow between subjects and objects with different MAC labels. We consider that X resources are objects, and thus subject to MAC. In TX, every object (e.g., window, pixmap) which is created is given the MAC label of the subject which created it.

TMach implements a Bell-LaPadula policy which allows for read-down and write-up.<sup>10</sup> However, write-up is not feasible for TX, because it creates a high-bandwidth covert channel.<sup>11</sup> Read-down is not generally a problem in TX, although there is at least one protocol request to read an object which can cause an event to be sent to the owner of the object, thus providing a covert channel.

In TMach, MAC is normally provided by the Name Server. TMach tasks ask the Name Server for access to an object, and if MAC (and DAC) are satisfied, then the Name Server provides a port to the object manager. For TX, this is not feasible for several reasons. First, every protocol request would have to be sent to the Name Server (rather than to the X server) since the protocol requests reference resources by ID. This would mean redesigning the X protocol to use the Name Server protocol. Second, the performance impact of having every protocol request mediated by the Name Server would be enormous.

For these reasons, MAC in TX is not compatible with MAC in TMach, although it is consistent. That is, TX enforces a policy more restrictive than the TMach policy (e.g., the TX policy does not allow read-down or write-up, even though the TMach policy does).

## 5.3 Discretionary Access Control

Discretionary access control is required in some form by all systems at C2 and above, with the requirements including access control lists at B3. Within our research group, there was significant disagreement over what role (if any) DAC plays in a windowing

<sup>10</sup>Read-down allows a subject to examine a subject or object at a lower sensitivity level, while write-up allows a subject to change an object at a higher sensitivity level. The inverse operations, read-up and write-down are prohibited by Bell-LaPadula.

<sup>11</sup>If a client writes to a higher level object, it must either receive an error that the object did not exist, or it does not receive an error in which case the client knows that the object does exist. Since clients pick their own resource IDs, a high level client could signal to a low level client by selectively creating certain resource IDs. This creates a covert channel.

system.

If a given X display were only to be used by a single user at a time (i.e., all clients belonged to that user), then DAC would not be needed. However, X encourages cooperation, allowing one user to create a window on another user's screen. While MAC provides safeguards against inadvertent disclosure of information, we felt that judicious use of DAC would allow for greater safety when sharing a screen.

In file systems, DAC is typically a few permissions (e.g., read, write, execute in UNIX). In X, we found that a larger set of permissions was required (currently eight), with an access control list.

As with MAC, TMach normally provides DAC through the Name Server. The same problems exist with providing DAC in the Name Server as providing MAC. For these reasons, DAC in TX is not compatible with DAC in TMach, although it is consistent. As with MAC, the TX DAC policy is more restrictive than TMach's DAC policy.

## 5.4 Privilege

Most systems provide some notion of privilege. For UNIX systems, this is "root," which has the power to do anything. TCSEC B3 systems require a notion of *least privilege*, which requires a finer granularity than the all-encompassing "root."

As discussed above, X has no notion of privilege. However, many operations (such as destroying another client's windows or modifying the keyboard mapping) need to have some method to allow the operation, while providing some restrictions. TX defines 12 privileges which can be associated with tasks.

TMach provides privileges for its own use, but does not have any mechanism for application defined privileges. For TX, there must be a method to associate TX privileges with clients, and for the server to know what TX privileges the client has, given a connection to the client. The client cannot be trusted to tell the server itself; rather, the privilege set must come from somewhere inside the operating system. It is not clear yet how this will be accomplished in TX.

## 5.5 Audit

Trusted systems are required to audit security relevant events. In a windowing system, it is not clear what is security relevant. For example, the TCSEC specifically requires auditing when objects are opened and closed. However, X has no concept of opening or closing an object. Instead, we feel that creating and destroying objects should be audited, along with attempts to violate the security policy and attempts

to exploit known covert channels.

## 5.6 Font Management

Font management is a special case in X. It is one of the few instances where the X server opens a file.<sup>12</sup> Font management also presents a unique problem: X programs frequently use fonts to represent pictures (because fonts are a fast way to load images). Therefore, fonts could have different MAC labels (e.g., a secret font might contain symbology for military hardware), and could also have different DAC. Thus, when a client asks for access to a font, the server must open the font file with the identity and attributes of the client, and not using the server's identity and attributes. Fortunately, the TMach Name Server provides a mechanism to request access to an object as another user.

## 5.7 Missing Facilities in the Current TMach Prototype

We have identified three areas where the current TMach prototype is missing needed functionality: application privileges, network support, and pseudo-terminals.

As discussed previously, some notion of application privilege is required for trusted X. For our prototype, we will allow clients to "claim" whatever privileges they need. However, for a useful trusted X system, this facility must be provided by TMach.

The TMach prototype is designed for use as a stand-alone system. However, the power of X comes from its ability to operate in a distributed environment. For a trusted X system to be useful, TMach will need extensions to support a trusted multi-level network.

The TMach prototype does not support pseudo-terminals (ptys). While this is not a problem for the TX server, some of the most common X clients (such as terminal emulators) rely on ptys. For a trusted X system to be useful, some notion of ptys is needed. Each pty can be single level, but it must be possible to create ptys at arbitrary MAC labels.

## 6 Architecture of the Trusted X Server

This section describes the architectural changes to the X server to meet B3 minimality and modularity requirements. It also describes the TX trusted shell, use of

---

<sup>12</sup>The only other instance is to open the color database.



TMach services, and the window manager. Only those architectural changes which impact TMach are discussed.

## 6.1 Structure of the Trusted X Server

In order to meet the requirements described above, we examined the architecture of the X server to determine required changes. As a major requirement of B3 systems is minimality and modularity, we also examined the server for alternate organizations which might allow portions of the server to be untrusted.

We divided the X server into three portions, as shown in Figure 2. The mediator receives protocol requests from clients and determines whether they meet the MAC and DAC policies. If so, it passes the request on to the appropriate drawing task. The drawing task performs the request, and sends the response (if any) to the mediator and the window image to the display manager. The display manager takes the window images provided by all of the drawing tasks and places them on the screen, managing the stacking order.<sup>13</sup> The display manager also manages the keyboard and pointer (mouse), sending events as appropriate to the mediator to be forwarded to the clients. The TX server contains one mediator task, one drawing task per MAC label, and one display manager task.

To meet minimality requirements, we initially planned for the drawing tasks to be untrusted. By using separate tasks (rather than threads), data from different MAC labels would never be mixed. However, we determined that the drawing tasks must be trusted, because they must not violate the DAC policy. That is, although each drawing task only contains data at a single MAC label, the data may have different DAC values. While the mediator is responsible for determining whether the protocol request is acceptable, the drawing tasks must be trusted not to access resources other than those named in the protocol request.

One of the limitations of this scheme is that certain data will be replicated among the various drawing tasks. For example, font images will be read by each drawing task. As a future enhancement, we may build a font manager task which will manage the fonts used by all of the drawing tasks. The font manager will then pass the fonts to the drawing tasks as TMach messages.

Communication between the different tasks (mediator, drawing, and display manager) uses TMach ports. TMach ports differ from those in Mach since they are labeled with a security level. The label of a port must dominate the level of the messages it contains. The "send" and "receive" services on ports are mediated by the TMach kernel. In addition, the transfer of send and receive rights between tasks

---

<sup>13</sup>Children of the same parent window may stack on top of each other. This relationship is known as the stacking order.

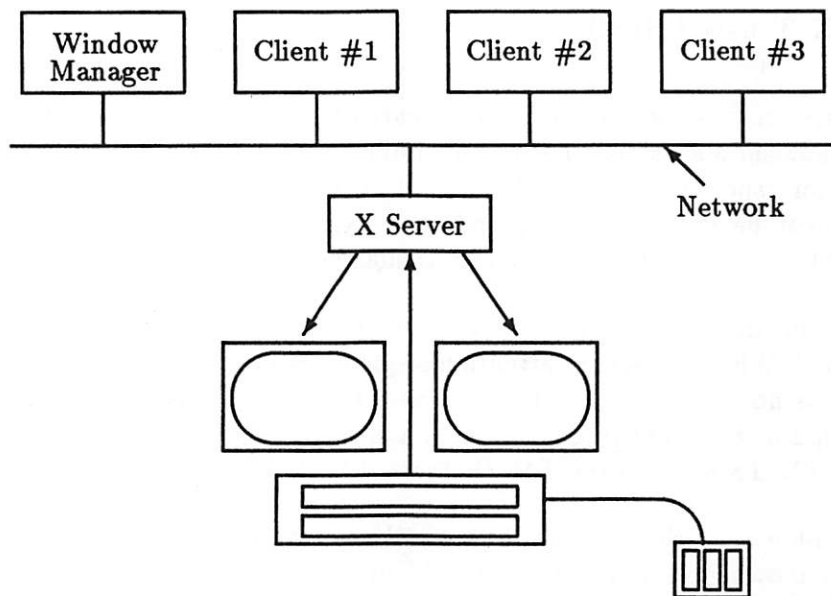


Figure 2: X Architecture

is mediated by the TMach kernel.

We expect to use the TMach messaging capability quite heavily. In particular, the notion that messages are not copied, but rather the memory is made available copy-on-write will be quite helpful, as it will improve performance. For example, the display manager will not need to write into messages it receives, so no copying will be performed.

## 6.2 TX Trusted Shell

Trusted systems are required to have a *secure attention* mechanism at B2 and above. The mechanism allows the user to communicate directly with the system without allowing for “spoofing” by malicious applications. Thus, the secure attention processing must be provided in the operating system. Secure attention is typically implemented in secure systems as a particular key or key sequence.

In TX, a particular keyboard/pointer combination will be used to invoke secure attention.<sup>14</sup> When the secure attention sequence is invoked, the trusted shell client (TX/TS) is notified. TX/TS allows the user to perform sensitive functions, such as logging in and out, changing passwords, and examining the MAC label of a client. As such, TX/TS replaces the TMach Trusted Shell.

Several minor extensions to the X protocol are required to support TX/TS. These consist of allowing a privileged client to identify itself as the trusted shell client, and a new event to notify TX/TS that the user invoked the secure attention sequence.

## 6.3 Trusted Window Managers

Since X allows for a variety of window managers which provide different look-and-feel to the user, we set a goal to allow the use of untrusted window managers with TX. This would allow the TX user the same choice of look and feel as a user of (untrusted) X. However, we determined that the window manager must manipulate windows of different MAC labels (e.g., to place them on the screen, iconize them, resize them, etc.). This requirement forced us to conclude that a trusted window manager is needed.

OSF/Motif is the most popular look-and-feel in the X community today, and is available at a relatively nominal cost. There being no overwhelming technical reasons

---

<sup>14</sup>We originally planned to have the user click on a reserved area of the screen, but discovered that the X protocol allows clients to lock the pointer into a particular window, thus preventing motion to the reserved area.

to choose one window manager over another, we have selected Motif as the basis for our trusted window manager, which will be known as TX/Motif.

Unlike the server, we do not expect to restructure Motif into trusted and untrusted components. Feasibility of such a restructuring may be examined in the future.

The window manager has several interactions with the operating system. For example, it reads configuration files and starts clients. A trusted window manager could also be used to allow users to interactively start clients at different MAC labels. TX/Motif will require changes to use TMach Name Server and File Server requests (rather than UNIX primitives) to perform these functions.

## 7 Related Work

Others looking at trusted X include the Compartmented Mode Workstation (CMW) community (Mitre, Sun, DEC, SecureWare, IBM, and Addamax). CMW requirements [15] are generally a superset of TCSEC B1 requirements.

The major differences between our work and the CMW work are the higher level of trust (B3 vs. B1) and the base operating system (UNIX for CMWs). These differences have led us to a very different architecture for TX than that used by CMWs.

## 8 Conclusions

Our analysis of X concluded that a B3 implementation is possible, given appropriate restructuring of the server and restrictions on the protocol. We found some areas of TMach (the servers, ports, messages) which we expect will match our requirements well. Areas which caused difficulty include DAC and privileges.

We recently completed the preliminary design and analysis phase of our project. A formal model has been developed to describe our security policy, and a lower level document have been written to describe the security semantics of each protocol request. Other related documents have been written, and a prototype has been developed to examine visible labeling policies. Over the next year we plan to implement significant portions of the TX server and the TX/Motif window manager as prototypes.

## 9 Acknowledgements

The architecture of the TX server was cooperatively developed by a team which included both authors and Hilarie Orman and Glenn Benson of Trusted Information Systems and John McHugh and Charlie Martin of Computational Logic, Inc.

## References

- [1] Morrie Gasser, *Building a Secure Computer System*, Van Nostrand Reinhold, 1988.
- [2] Avadis Tevanian, Jr. and Richard Rashid, *MACH: A Basis for Future UNIX Development*, Carnegie Mellon University, June 1987.
- [3] Jeffery Graham and Wayne Morrison, *Trusted Mach System Architecture*, TIS Report #324, Trusted Information Systems, April 1990.
- [4] *X Window System Protocol*, MIT X Consortium Standard, X Version 11, Release 4, Robert Scheifler, 1988.
- [5] *Xlib — C Language Interface*, MIT X Consortium Standard, X Version 11, Release 4, James Gettys, Robert Scheifler, and Ron Newman, 1989.
- [6] *X Toolkit Intrinsics — C Language Interface*, MIT X Consortium Standard, X Version 11, Release 4, Paul Asente and Ralph Swick, 1988.
- [7] *X Athena Widget Set — C Language Interface*, MIT X Consortium Standard, X Version 11, Release 4, Chris Peterson, 1989.
- [8] *OSF/Motif Programmer's Reference*, Open Software Foundation, Prentice-Hall, 1990.
- [9] Jennifer Steiner, Clifford Newman, and Jeffrey Schiller, "Kerberos: An Authentication Service for Open Network Systems" in *Proceedings of the Winter USENIX 1988 Conference*.
- [10] *Inter-Client Communication Conventions Manual*, Version 1.0, MIT X Consortium Standard, 1989.
- [11] Susan Angebrannt, Raymond Drewry, Philip Karlton, and Todd Newman, *Definition of the Porting Layer for the X v11 Sample Server*, 1988.
- [12] Susan Angebrannt, Raymond Drewry, Philip Karlton, and Todd Newman, *Strategies for Porting the X v11 Sample Server*, 1988.
- [13] *X Display Manager Control Protocol*, MIT X Consortium Standard, Version 1.0, 1989.



- [14] National Computer Security Center, Fort Meade, MD, *Trusted Computer Systems Evaluation Criteria*, DoD 5200.28-STD, December 1985.
- [15] *Security Requirements for System High and Compartmented Mode Workstations*, DIA Document Number DDS-2600-5502-87, John P. L. Woodward (Mitre), November 1987.

...the ...  
...the ...  
...the ...  
...the ...  
...the ...

# Building a Fault-Tolerant System Based on Mach

*Rong Chen and Tony P. Ng*

Department of Computer Science  
University of Illinois at Urbana-Champaign  
Urbana, IL 61801

## Abstract

In this paper, we propose a fault-tolerant system prototype based on the Mach. In our system, all processes are structured as clients/servers. Every process may checkpoint its state to a different machine at its own pace. However, if a client is sending out a message that will mutate the state of a server process, it is forced to checkpoint first; otherwise, the client sends out the message directly. With carefully designed server interfaces, client processes can achieve fault-tolerant computing transparently. The overhead incurred due to checkpointing can be further reduced through data caching and by using the Mach external memory management mechanism.

## 1 Introduction

With the declining prices of computers and networks, a distributed computing environment can not only facilitate an economical way of increasing MIPS (million of instructions per second), but also offer a practical way of increasing reliability. It is common practice nowadays to use several workstations, each engaging in a different part of the same job, to achieve higher system throughput. Yet, it is not so common to use several workstations, cooperating on the same part of the same job, to prevent system crashes in the event of failure of an individual workstation. We believe the latter can be achieved without paying a very high price or requiring any special purpose hardware devices.

In this paper, we propose a fault-tolerant system prototype based on the Mach operating system [1, 2]. In our system, every process may checkpoint its state to a different machine at its own pace. However, if a process is sending out a message that will mutate the state of a receiving process, it is forced to checkpoint first. The information as to whether a message would change the state of receiving process is defined by using the modified MIG [3] interface specification language. We call the communication interface generated by MIG, a *contract*. A message being sent through a contract will guarantee not to cause any inconsistency between the sending process and the receiving process despite failures of the sender, the receiver, or both. Therefore, the consistency of global states is guaranteed at all times.

Our proposed system is an otherwise regular operating system (i.e., Mach) except for those processes that have to be resilient to failures. The main advantage of using contracts is that checkpointing can be avoided when a message does not mutate the receiver's state. In our computation model, we do not assume repeatable process executions, that is, we allow non-deterministic executions. Messages are not required to arrive in the same order when a program is executed twice. It is, however, the responsibility of the designer of the server (receiver process) to eliminate all non-deterministic program behavior by carefully defining the contract between the server and its clients (sender processes). For example, if a RPC message could potentially be non-deterministic, the sender should checkpoint before the request can be processed by the receiver. With our simple checkpoint protocol, both multiple rollbacks and cascading rollbacks are prevented.

We are implementing our proposed fault-tolerant system to verify its practicality. In this paper, we will demonstrate through a file server example that the number of checkpoints may be decreased substantially by taking advantage of the Mach external memory management facilities. Our intermediate goal is to have several Sun workstations, connected via a local area network, running concurrently to withstand hard failures. We will also demonstrate the performance of such a system and evaluate the runtime overhead of several applications. The next phase is to investigate other ways in which semantics of applications can be used to reduce the cost of achieving reliability.

In the next section, we define the problems associated with fault-tolerant computing and give some basic assumptions. Our checkpoint protocol are presented in section 3. Section 4 describes some optimization strategies. We dedicate section 5 to details of the implementation of our fault-tolerant system. In section 6, we survey related works in this area. Section 7 summarizes this paper.

## 2 The Problem and Assumptions

A distributed system is called *reliable* or *fault-tolerant* if it has the capability to allow user-level processes to survive system failures, including hard and transient failures. A process that can withstand system failures is called a *resilient* process. The choice as to whether a particular process is to be executed as a resilient process is made by the user when that process is started. For any resilient process (also referred to as *primary* process), we assume that there is a backup process, presumably on a different workstation. When the primary fails, the backup will resume the computation.

Resilient processes are inherently more complex than their conventional non-redundant counterparts. In addition to the usual concerns about functional correctness, programmers must now address issues arising from concurrency control (e.g., the synchronization of a primary process with its backup), and fault-tolerance (e.g., failure detection, process re-

sumption, etc.). The core problem of fault-tolerant computing research is how to guarantee that the states of resilient processes are always consistent [4], regardless of failure. Resilient processes usually suffer from limitations such as deterministic execution [5] and cascading rollbacks [6].

In our computation model, a process can be in only one state at a time. The state of a process may change to another state after the process receives a message. The state change is irreversible. is performed whenever the state changes. The states of all communicating processes, at a given time, are *consistent states* if they are reachable through normal executions. In our system, we assume that the processes fail by stopping, that is, the computation would not continue if failures happen. We also assume that a primary machine and its backup machine(s) do not fail at the same time. Another assumption that we make is that when a process crashes, it forgets what it has done from its last checkpoint.

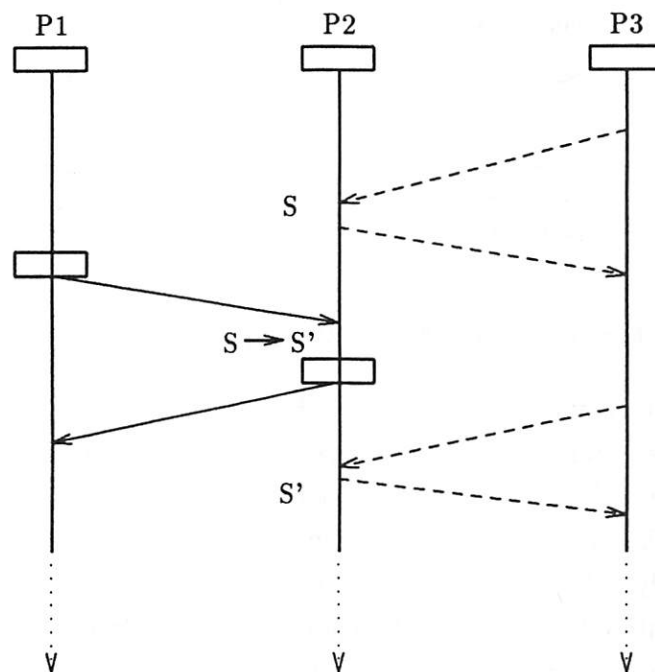


Figure 1: three executing processes

Now, consider three processes, P1, P2 and P3, executing simultaneously (Figure 1). P1 sends a message to P2 (represented by a solid arrow) and P2 responds by changing its state from S to S'. P3 sends two messages to P2 (depicted by dashed arrows) and neither of these messages changes P2's state. P2 sends two replies back to P3 (also depicted by dashed arrows). If we checkpoint at the boxed locations, any one of the processes can be resilient to failures in the sense that it can always restart from its last checkpoint independent of others. Our message passing scheme is based on Mach IPC mechanism. A message in solid arrow will be resent after the sender detects a failure of the receiver; duplicate messages



will be eliminated. Different kinds of messages are distinguished by the contracts (MIG interface) through which the messages are being sent. Assigning semantics to messages makes it possible to optimize the checkpoint algorithms. We will discuss this in the next few sections.

### 3 Checkpointing Protocol

Our fault-tolerant system is a backward recovery system that preserves consistency without cascading rollbacks. To avoid checkpointing at every message, we treat different messages differently, depending on the semantics of messages. The checkpoint protocol guides the system in deciding which messages should be checkpointed.

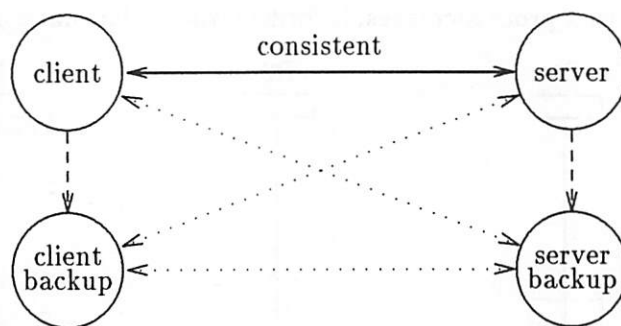


Figure 2: consistency between primaries and backups

Consider two communicating processes. For convenience, we label them as server and client processes (see Figure 2). Furthermore, suppose there are two backup processes in a backward recovery system, i.e., a server backup and a client backup, respectively. A backup process receives checkpoint information from its primary periodically, when the primary is executing normally. When the primary fails, the backup process becomes the primary and resumes the execution from the last checkpoint it received from the old primary. The state of a backup process is the state of its primary at its last checkpoint. In order to guarantee consistency when one or both primaries fail, the state of the backup process must also be consistent with the state of the other primary or the backup of the other primary; this is indicated by the dotted lines in Figure 2.

To guarantee that a backup process is consistent with the other primary or its backup, we propose a simple checkpoint protocol for our fault-tolerant system. Suppose we represent the client process and the server process as  $P_1$  and  $P_2$ , respectively. Assume that the current state for  $P_1$  is  $s_1$  and for  $P_2$  is  $s_2$ . Let  $\delta_2$  be the state transition function of  $P_2$  that maps  $P_2$  from one state to another upon receiving a message. Let  $m$  be a message sent from  $P_1$  to  $P_2$ . The checkpoint protocol for  $P_1$  would be invoked when  $m$  is sent:

```
if ( $s_2 \neq \delta_2(s_2, m)$ ) checkpoint( $s_1$ ); else skip.
```

The inequality defined by  $P_2$  indicates that the message  $m$  would change the state of  $P_2$ , and the new state  $\delta_2(s_2, m)$  is different from  $s_2$ . If the inequality is true, we say that  $m$  mutates the state of  $P_2$  and  $P_1$  has to checkpoint before actually sending out the message. If the predicate is false, we say that  $m$  is *side-effect-free* and  $P_1$  does not have to do anything special while sending out message  $m$ . When  $P_2$  has finished processing a message  $m$  that changed its state, it sends back a reply to  $P_1$ . This reply message is treated the same way as a message from  $P_2$  to  $P_1$  that will mutate  $P_1$ 's state. That is,  $s_2$  will be checkpointed since  $P_2$  assumes  $P_1$  has changed its state implicitly.

The checkpoint protocol guarantees a invariant for the whole system: the last checkpoint state of a process  $P$  is always consistent with the current states of its communicating processes at any time. The validity of this invariant can be argued informally as follows: since the messages sent by  $P$  after its last checkpoint are side-effects-free, a receiving process could not have modified its state that reflect the fact that the messages had been sent at all. Messages that have been received by  $P$  after its last checkpoint can be treated as lost messages. Given a set of communicating processes, the states of these processes would remain consistent after an arbitrary subset of these processes are rolled back to their last checkpoints. This can be shown to be true by applying the invariant repeatedly.

## 4 Optimization Strategies

Using our checkpoint protocol, the cost of checkpointing depends on the interface between the server and the client in an application. By changing the interface, an application can decrease the frequency of checkpointing and therefore, the cost of checkpointing. For instance, in a file server supporting sequential file access, an application has to keep track of an index which points to the next location for file access. If this piece of information is kept in the server, the client would only have to issue an access request and the server would be able to determine the location from its own state. According to our checkpoint protocol, since the state of the server (the index) is modified each time by an access request, the client is forced to checkpoint each time it accesses a file. Similarly, the server is forced to checkpoint each time it returns from an access request in order to keep the consistency of the server state (the real index) and the client state (the assumed index). On the other hand, if the index state is kept by the client, the index would have to be sent along with each access request, but no checkpointing would be needed by the client or by the server if the access is a read request.

Therefore, the modified file server interface for a sequential file system such as Unix would look like that in Figure 3. Upon receiving a message, the contract dictates that one

open	checkpoint
read	pass
write	checkpoint
close	checkpoint

Figure 3: contract of a conventional file server

of the two actions, *pass*, or *checkpoint* be performed. When a file is opened, a checkpoint is performed in order to guarantee the consistency between the server and the client; since reading a file block does not change the state of file server, the message is simply passed through. Whereas, every block written is subject to later reading, so write messages have to be checkpointed right away. Checkpointing every block written is expensive, yet it guarantees strict UNIX file semantics. However, most distributed file systems (e.g., SUN NFS, AT&T RFS and CMU AFS) cache data on the client machine, avoiding strict UNIX emulation for efficiency reasons.

The concept of *memory-mapped-file* is studied in Mach [7]. As a file is opened, the whole file is mapped onto the local virtual memory space. Subsequent read and write calls are treated as accesses to the local memory. The memory-mapped-file concept changes the semantics of UNIX sequential file access to random access. The original concern of memory-mapped-file is to improve the efficiency of distributed file servers. If we take a closer look at the implementation of a memory-mapped-file, we will see two copies of the same file, one on the client machine and the other on the server machine. We feel that it is possible to take advantage of the memory-mapped-file concept and use it to improve the reliability of both the server and the client. Specifically, the copy mapped on the client machine can be treated as a backup to the file on the server. If the server fails, the client can flush its local copy to the server backup and rebuild the server state.

Under the new scenario, a memory-mapped-file need only be checkpointed when the file is opened or closed. When a file is opened for writing or updating, a new version number is assigned to the client copy of the file. When the file is closed, the whole memory-mapped-file will be flushed back to the server. This paradigm is similar to the cache consistency and disk spooling problems in computer architecture design. Intermediate write operations can be sent back to the server, but they are merely treated as optimizations to the general mechanism. No checkpoint is required for write operations. The close operation will flush back the dirty pages that have not been sent to the server or flush all the modified pages to the server depending on whether the primary server has failed after the last open operation. The performance improvement over the sequential file access mechanism is obvious.

Based on application-specific knowledge, a user may reduce the cost of checkpointing and recovery even further. In the X window system [8], redrawing the contents of a window is the responsibility of a client (application) program. A client program has to handle the redraw signal explicitly when a window is resized or moved to the top level (i.e., exposed on the screen). Keeping reliability in mind, the ability to redraw a window is almost exactly what is needed when a failed window manager recovers. For instance, if a window manager fails, we can always start another one and send the X application a redraw message hoping that it will reprint what was lost on the screen.

In our research, we feel that the memory-mapped-file paradigm (to be more precise, the Mach external memory management paradigm) deserve more study. Data caching improves not only efficiency but reliability as well. Many application softwares may be redesigned to use the paradigm [2]. We shall also investigate how this kind of application-dependent recovery can be supported in our system. For example, we can design a simple recovery protocol that requires a client to handle a special message, *recall*, explicitly. After a server recovered from a failure, a recall message is sent to the client from the kernel, and the client can flush its backup buffers to the server to keep its information up to date.

## 5 The Checkpoint System

This section presents details on how to build the proposed system, which is based on the Mach Operating System developed at CMU.

We anticipate that there could be two ways to start a resilient process. First, a user may indicate that a binary file should be executed as a resilient process when the file is executed. Alternately, a user may choose to mark a file statically with a flag to indicate that the file should be executed as a resilient process whenever it is invoked. We plan to support both methods.

In Figure 4, the components of the checkpoint system are shown. When the user process starts, the kernel informs the checkpoint manager to spawn a daemon process, called the *checkpoint pager*. The checkpoint pager is responsible for checkpointing the user process and for failure detection. There is one checkpoint pager for each resilient user process, but there is only one checkpoint manager for each machine.

### 5.1 Checkpoint Manager

After a checkpoint manager is informed of a new user process, it tries to contact another checkpoint manager on a different machine (the backup machine). The checkpoint manager on the backup machine will in turn start a checkpoint backup server, which will communicate with the checkpoint pager in order to construct checkpoints on the backup machine. When

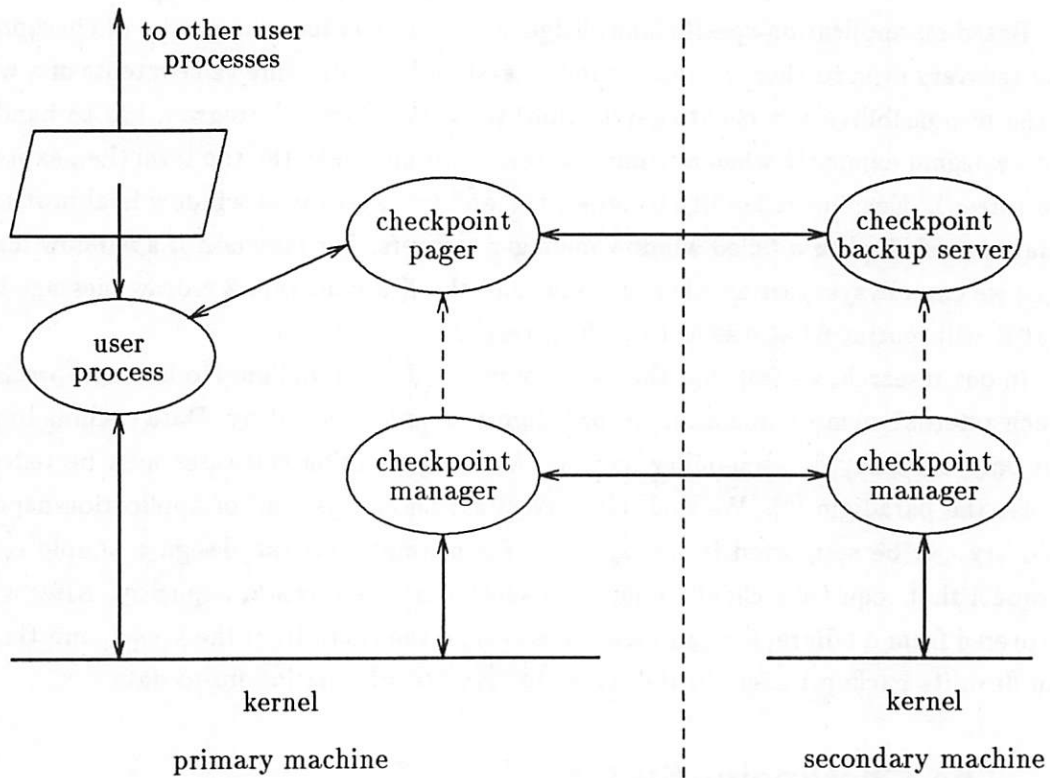


Figure 4: the checkpoint system

a port is created by the primary process, its backup port (ownership right) is assigned to the backup server. All messages for the primary process are re-routed to the secondary process by kernels of the message senders if the primary fails. Similarly, when a secondary process begins, another backup server might be started elsewhere.

## 5.2 Checkpoint Pager

The process can be checkpointed either when the kernel reschedules it or when a checkpointing function is invoked. When checkpointing starts, the user process is suspended. The checkpoint pager will find all dirty pages of the user process and mark them read-only. After this is done, the user process is resumed immediately. The checkpoint pager executes asynchronously with the user process. The first request from the pager is sent to the checkpoint manager of its backup machine and the manager starts a backup server for the resilient process on its local machine. Finally, all dirty pages are sent to the backup machine and their access permissions are changed back to read-write. If the user process writes to one of the dirty pages that has not been saved, that page will be sent to the backup machine with priority.



Checkpointing a Mach process is simpler (at least in theory) than checkpointing a process of a more conventional operating system, such as Unix, because all system objects in Mach are referenced through a single system object abstraction, i.e., port. When a backup process resumes execution at the last checkpoint on a different machine from the primary's, it needs only to relocate all the ports associated with the resilient process. In Unix, different system objects (e.g., socket number, file number, pid, etc.) have to be treated differently.

### 5.3 Checkpointing by External Pager

Mach virtual memory may have some segments managed by external pagers, as shown in Figure 5. We assume P1 is the default memory pager; P2 and P3 are two external pagers. The dirty pages in M1 is dumped by the checkpoint pager to the backup server. The dirty pages in M2 and M3 should be dumped by P2 and P3, respectively. In other words, checkpointing a resilient process in this case is subject to three sub-checkpoints. The three sub-checkpoints should be performed atomically, which is guaranteed with a two-phase commit protocol. In the first phase, a checkpoint pager requests all external pagers to take tentative checkpoints. Depending on whether all tentative checkpoints are successful, the checkpoint pager decides whether to commit or to abort the checkpoints. In the second phase, the checkpoint pager's decision is carried out by all processes.

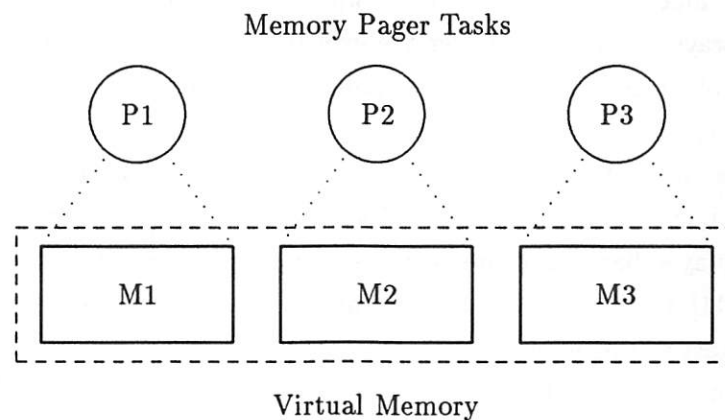


Figure 5: checkpointing with external pagers

### 5.4 Backup Server

A backup server is the secondary process for a resilient process. Its duty is to save an executable image of its client on disk as a regular binary file in *macho* format [9]. Another important duty of a backup server is to timeout a dying resilient process. The timeout interval is given by the user at start-up time, or is chosen by default. In case the primary

user process fails, the backup server will turn itself into a secondary user process and continue the computation.

## 5.5 The Contract

A contract is produced by the MIG interface generating language. By adding a key word, either *const* or *mutate*, in front of a procedure or function description, the MIG program indicates those requests (messages) which may be sent by the client and which will change the server state. That piece of information is translated into two stub programs, one for the server and one for the client. If a procedure or function is prefixed with *const*, the message is sent as usual. Otherwise, a checkpoint routine and a message resending routine are inserted in the client code; in the server code, a duplicate message detection routine and a checkpoint routine are put in appropriate places.

## 6 Related Work

In this section, we review a few specific systems that have been proposed or implemented. The Targon/32 system [10] is a general purpose UNIX system with special bus hardware that guarantees three-way atomic message transmission. Since all interprocess messages are delivered to the inactive (or stand by) backups for the sender and receiver processes, both backups are always ready to take over the execution from their primaries. Targon system provides fault-tolerance of single process failures under the assumption that all processes are deterministic.

Following the idea of Targon/32, Babaoğlu has suggested a pure software solution [11] based on the Mach operating system. When a process is restarted, previously received messages are “played back” and previously sent messages are discarded. As the author has acknowledged [11], three-way message transmission is a challenge, given the current message passing (broadcasting) technology.

Both systems proposed by [5, 12] save messages in volatile memory on the sending machine, rather than using the receiving machine or a third backup machine for this purpose. Johnson et al. assume that at most one machine may fail at any time. Whereas, the system proposed by Strom et al. guarantees that all processes have consistent checkpoints but it does not prevent multiple rollbacks. Koo and Toueg [6] have proposed a two-phase commit checkpoint algorithm to guarantee that a set of checkpoints are consistent.

The *optimistic* checkpoint strategy assumes that errors are infrequent. It allows checkpointing or message logging to be performed asynchronously with computations [13]. When an error is detected, the resilient process state will be *reconstructed* using the last checkpoint and saved messages. On the other hand, in the *pessimistic* checkpoint strategy, a resilient

process will not continue its execution unless a checkpoint or a message backup has been made securely.

An experimental fault recovery mechanism based on UNIX has been implemented for IOSYS by D. Taylor [14]. He takes advantage of the UNIX system call *fork* to generate process images as checkpoints. When a failure occurs, either transient or hard crash, the system will resume the resilient process from its last forked image. Apparently, a mechanism of dumping only the dirty pages is more efficient, but Taylor has chosen to avoid the complexity of modifying the UNIX kernel. Disk files are handled separately with system assistance or with user guidance.

## 7 Conclusions and Summary

In this paper we proposed a backward recovery fault-tolerant system. Using such a system, we can investigate how to write application software to facilitate resilient computing. We also expect to implement the proposed system to show its practicality. Building a system will provide us with the performance evidence to verify the potentials and limitations in the area of resilient computing.

Our computation model is based on a message-passing operating system paradigm. All processes are structured as clients/servers. The client/server interface generated by MIG is augmented with checkpoint information. We call such an interface a *contract*. The checkpoint information is used by the operating system to enforce a checkpoint protocol at run time when processes communicate. The checkpoint protocol guarantees that any two communicating processes are consistent at all times despite failures. The purpose of introducing the contract into our system is to capture the semantics of messages and to use it to reduce the frequency of checkpoints.

The performance gain attainable by using the semantics of messages, though important, is not yet known. With carefully designed server interfaces, client processes can achieve fault-tolerant computing transparently. The overhead incurred due to checkpointing can be further reduced through data caching and by using the Mach external memory management mechanism. Checkpointing a Mach process is simpler than checkpointing a process of a more conventional operating system, such as Unix, because all system objects in Mach are referenced through a single system object abstraction — port.

## References

- [1] Richard Rashid et al. Mach: a foundation for open systems. In *Proceedings of the Second Workshop on Workstation Operating Systems*, pages 109–113, IEEE, Pacific Grove, California, September 1989.

- [2] Michael Young et al. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th Symposium on Operating Systems Principles*, ACM, November 1987.
- [3] Richard P. Draves et al. *MIG — The Mach Interface Generator*. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, November 1989.
- [4] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Prentice Hall, Englewood Cliffs, New Jersey, 1981.
- [5] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, pages 14–19, Pittsburgh, Pennsylvania, July 1987.
- [6] Richard Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.
- [7] Avadis Tevanian, Jr. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach*. Technical Report CMU-CS-88-106, School of Computer Science, Carnegie Mellon University, Pittsburgh, July 1987. Ph.D. Thesis.
- [8] Robert W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.
- [9] Avadis Tevanian, Jr. et al. *A Unix Interface for Shared Memory and Memory Mapped Files Under Mach*. Technical Report, School of Computer Science, Carnegie Mellon University, Pittsburgh, July 1987.
- [10] Anita Borg, Wolfgang Blau, et al. Fault tolerance under Unix. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.
- [11] Özalp Babaoğlu. Fault-tolerant computing based on Mach. *Operating Systems Review*, 24(1):27–39, January 1990.
- [12] Robert E. Strom, David F. Bacon, and Shaula A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, pages 44–49, Tokyo, Japan, June 1988.
- [13] Robert E. Strom and Shaula A. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.
- [14] David J. Taylor. Backward error recovery in a Unix environment. In *Proceedings of the 16th Int'l Conf. on Fault-tolerant Computing Systems*, pages 118–123, Vienna, Austria, July 1986.

# Transparent Recovery of Mach Applications

Arthur Goldberg\*    Ajei Gopal†    Kong Li‡    Rob Strom§    David F. Bacon¶

## Abstract

We have built a software layer on top of Mach 2.5 that recovers multi-task Mach applications from fail-stop failures. The layer implements Optimistic Recovery (OR), a mechanism for transparent recovery from failing tasks and processors, based on asynchronous checkpointing and logging of inter-task messages. OR recovers from failure by restoring a checkpoint and replaying the logged messages.

The current prototype supports message communication via sends and receives, simple port operations, and task interactions through the environment manager.

This paper discusses the issues Mach raised for this implementation, the structure of the OR layer, the design of future enhancements, and comparisons with other recovery techniques.

## 1 Introduction

During the lifetime of a distributed system one or more of its tasks may fail. Rather than abort the entire system due to a single failure, it is preferable to make the computation fault-tolerant—that is, the computation should continue to function correctly despite the failure of processors or individual tasks.

There are two approaches to fault-tolerance. Either fault-tolerance can be explicitly written as part of an application, or it can be written as a general purpose function available to many applications. The second approach—the one we choose—is based on transparent fault-tolerance.

\*IBM T.J.Watson

†Cornell University. This research was begun when this author was visiting IBM. Also partially supported by an IBM Graduate Fellowship.

‡UCLA. This research was begun when this author was visiting IBM.

§IBM T.J.Watson

¶UC Berkeley

Transparency implies two things. First, to make an application recoverable, no application code must be changed—at worst, the application may have to be recompiled or relinked. Second, the fault-tolerant system must produce the same output<sup>1</sup>, even when failures occur, as the original non-recoverable system.

We have built a layer on top of Mach [15, 21] that makes multi-task Mach applications transparently recoverable from fail-stop processor failures. The layer implements Optimistic Recovery (OR) ([18]), an ‘optimistic’ mechanism for fault-tolerance.

A transparent recovery mechanism for Mach applications is desirable for several reasons. First, Mach is oriented towards distributed systems, so Mach applications are likely to be multi-task programs that run on multiple machines. Such applications are ideally suited for recovery since they suffer the failure of *some* component more frequently than a program running on a single machine.

Second, Mach applications are likely to be long-lived systems like computation and communication servers which users rely on, and therefore need to be automatically recovered.

We choose to build OR on top of Mach, rather than another operating system, for several reasons. First, while not a perfect match, the Mach interface does present the basic communication primitives of the OR model. Second, Mach is a young operating system with a small community of users, which we hope to impact with our experience. We hope that future implementations of the OR layer can be more deeply incorporated into Mach, so that applications can be made recoverable as a routine matter.

This paper describes the design and implementation of the optimistic recovery layer, of which we have a running prototype (called Optimistically Recoverable Mach or ORM). We describe the particular challenges posed by Mach and how we overcame them. In some cases, we suggest alternative or additional facilities to

<sup>1</sup>Failures may cause stuttering—see Section 5.4.1.



make solving these problems easier, particularly where the facilities we require are likely to be required by other services layered on top of Mach, such as distributed debugging and process migration.

## 2 Introduction to Optimistic Recovery

Optimistic Recovery is an efficient technique for recovering a recent consistent state of a distributed system after the failure of any number of system components. We will present an abstract system model and will then precisely define what we mean by "failure", "recovery", "consistent", and "recent". We then summarize the algorithm.

### 2.1 Abstract Model

A distributed system or *logical machine* is composed of a number of *recovery units* (RUs) connected by one-way communications links. Each recovery unit consists of a processor, a local *volatile storage* used by programs running within the recovery unit, and a local *stable storage* used by the recovery mechanism. Computations running within an RU read and write volatile storage, and send and receive messages over the communication links. A communication link connects the recovery unit either with another recovery unit in the logical machine, or with the *outside world* external to the logical machine. We assume that writing to stable storage can be asynchronous and can overlap computation. The recovery mechanism initiates the transfer of data to stable storage, and later receives an acknowledgement that the data has been written.

A *failure* of an RU consists of the following events: (1) the processor halts—no further messages are sent or received, and no further writing to stable storage occurs; (2) the contents of volatile storage is lost; (3) messages in transit on links to or from the RU are lost; (4) sometime later the RU starts up again. This is a *fail-stop* failure[16]. In particular, a failure does not cause any corrupted data to be written to stable storage or corrupted messages to be sent.

A failure in a distributed system consists of the failure of one or more (possibly all) of its RUs.

A *correct execution* is a trace of external inputs and

outputs which would be observable in the absence of failure. Prior to a failure, the behavior of a system is some *prefix* of a correct execution. The state restored by the recovery mechanism after a failure is *consistent* if the future behavior is a *suffix* of that same correct execution.

Ideally, the behavior obtained by splicing the behavior prior to failure together with the behavior after recovery should be equal to the original failure-free execution. However, we cannot avoid a small amount of overlap or *stutter*—external inputs received in the recent past may be lost and may have to be resent; external outputs sent in the recent past may be delivered again. A consistent state is *recent* if there is a small, bounded amount of stutter. In particular, there is no *domino effect*—an unbounded cascade of rollbacks to the global start state.

A *transparent* recovery mechanism, such as OR, is an extension to the operating system in each recovery unit such that any application program which behaves correctly in the absence of failure will behave identically (except for stutter) in the presence of failure without having to be rewritten.

### 2.2 Overview of Algorithm

OR is described in detail in Strom and Yemini's paper[18]. Optimizations to reduce the cost of checkpointing and logging are described in other papers[17, 1, 19, 20].

OR is based on backwards error recovery[14]. A recent consistent state of a system is obtained by restoring recent states of each recovery unit such that for any communication link from RU *A* to RU *B*, (1) there are no *missing* messages—messages sent by *A* but not received by *B*, and (2) there are no *orphans*—messages received by *B* but not sent by *A*.

OR uses *checkpointing*, *logging*, and *replay* to restore states. Each recovery unit periodically takes a checkpoint by writing the state of its volatile memory to stable storage. Additionally, each input message processed by the recovery unit is logged to stable storage. We require each RU to be *piecewise deterministic*. This means that the state of any computation can be reconstructed by first restoring a prior checkpoint and then re-executing the computation using the logged messages rather than the communication links as the input source.

OR introduces a *session* protocol on each communication link. The sender *half-session* consecutively numbers each message, and saves a copy of messages not yet logged by the receiver. The receiver *half-session* checks for duplicate and missing messages. Duplicates arise as a result of replay; missing messages arise when a receiver fails after physically receiving a message but before logging it.

What makes OR “optimistic” is the fact that instead of avoiding orphans, we allow orphans to occur and subsequently roll them back. If *A* sends *B* a message while *A*’s current state is not recoverable because an earlier input message has not yet reached stable storage, the message may become an orphan, because due to a failure, that earlier message may never be logged. Rather than delaying the sender by requiring logs to be “force-written”, optimistic recovery instead tags each message with the identifiers of the unlogged antecedent messages and proceeds with sending the message, hoping that orphans are rare. These tags are called *dependency vectors*[18] or *dependency maps*. Each RU updates its local dependency map upon receiving a tagged message. The dependency map defines the set of unlogged messages on which the current state depends.

Most of the time no failure will occur—the antecedent message will eventually be logged, and a *log message* will be sent notifying other recovery units that the dependency tags may be removed from the dependency map. If all dependency tags are removed from a state or a message, the current state or message becomes *committable* and will never be rolled back. If a failure does occur, the dependent recovery units will be notified of which messages are lost. Any recovery unit containing a dependency on a lost message is known to have received an orphan message. The RU is rolled back to an earlier state which does not depend upon any lost message. All messages with receive sequence numbers greater than the sequence number of the last logged message before a failure are assumed lost. A new *incarnation* is begun starting at the next higher receive sequence number. Incarnation numbering is used to distinguish descendants of the message with this number from possible descendants of messages with the same receive sequence number in a previous incarnation.

The correctness requirement forbids us to send an uncommitted (potential orphan) message to the *outside world*. Otherwise one might observe a certain output prior to failure even though the restored state has a future inconsistent with the delivery of that output. Therefore, messages destined for the outside world are buffered by an *output boundary function* until all their

causal antecedents are logged.

### 3 Issues in Implementing ORM

Implementing ORM required dealing with several issues: (1) deciding how to map the abstractions of the OR model, such as recovery unit, log, and communications link, into specific Mach entities in ways which satisfy OR’s assumptions, such as piecewise determinism; (2) coping with features of Mach which make efficient implementation difficult.

We will summarize these issues separately:

#### 3.1 Mapping the Abstract OR Model

Certain abstractions map fairly obviously to physical reality. For example, the “volatile storage” of OR corresponds to virtual memory of Mach applications; “stable storage” corresponds to disk; a “failure” is a halt and reboot of Mach, e.g. after shutting off power.<sup>2</sup> Less obvious are the mappings for the concepts “recovery unit,” “communication,” and “logical machine.”

##### 3.1.1 Recovery Unit Granularity

We chose to map an RU to a Mach *task*. We thought this a good conservative decision for a first implementation. Each Mach task has an independent address space which can be viewed as the RU’s volatile memory. Mach tasks do not usually share memory. On the other hand, if we were to map an RU to a Mach thread, we would be faced with the problem of identifying which volatile memory belonged to which RU, and of mapping the shared-memory communication to message communication. Additionally, the smaller the RU, the more communication becomes inter-RU communication, which requires logging and dependency tracking. Finally, larger RU’s, e.g. clusters of tasks, are harder to make piecewise deterministic and would require enforcement of synchronization between tasks.

<sup>2</sup>For a recoverable system which is embedded within another system, even the notion of failure may admit multiple interpretations. For example, one can conceive of an implementation in which a single user’s program is a “logical machine”. The cancelling of the program by action of the system operator may be considered a “failure” in that environment. However, in this paper we will assume failures are hardware failures or power failures.

### 3.1.2 Communication

Communications links between RUs are Mach ports; OR messages are Mach "messages"; sending and receiving correspond to *msg\_send* and *msg\_receive*. For the most part the Mach model is a close fit to the OR model. However there are some differences, some of which necessitated careful design of ORM, others of which were temporarily ignored.

Mach ports differ from OR links in that in OR the sender knows the destination RU at the time a message is sent. In Mach, one has to do extra work to be able to determine the destination RU of a port for which one has send rights. Even then any such knowledge is only approximate because a port owner may pass receive rights at any time, even when the message is in transit. Additionally, after a failure, it is necessary to re-create ports, and Mach may assign different port identifiers. Similarly, after a failure Mach may assign a port identifier which happens to be identical to one which previously designated a different port. Because of these problems, it was necessary to implement the concept of a "reliable port". Reliable ports are to Mach ports as virtual addresses are to physical addresses. The applications programmer deals only with reliable ports; these ports never fail even when the communication link breaks or when recovery occurs. The recovery system tracks which Mach port corresponds to a particular reliable port. If no Mach port exists and one is needed, it is rebuilt.

In OR, there are two separate "receive" events: the physical arrival of a message, and the processing of the message by the program. Messages can be logged by the receiver as soon as they physically arrive. This can improve response time since if a message is logged sooner any external output which depends on the message can be released sooner. But there are two disadvantages: (1) we would have to dequeue the message from the Mach port in order to log it and then requeue it until the application program was ready to dequeue it; (2) if a message to port *x* arrives and is logged before another message to port *y*, it will be viewed as causally earlier, even if the program dequeues the second message first—this could result in a bogus dependency and an unnecessary delay in committing output. It is not clear whether it is better to log messages by arrival order or by dequeue order, but using dequeue order is easier to implement. Therefore in the current implementation, we pretend that messages are still in transit until they are actually dequeued.

When an application uses the Mach feature that associates message limits with ports, the mapping of OR "send" to Mach *msg\_send* is not straightforward. The success or failure of a *msg\_send* depends not only on the state of the sender, but also on the state of the receiver; therefore in principle dependencies flow from receiver to sender.

In the current implementation, we assume that all communication between tasks is via Mach messages. Mach actually provides several other means of communication. Tasks with appropriate rights can schedule other tasks. Memory can be shared between tasks as well as between threads within tasks. Finally, the current Mach implementation includes UNIX communication—signals, files, pipes, which eventually will be implemented using Mach, but which currently is not. Our current prototype ignores these complications, thereby restricting the scope of recovery. Our next plans call for the elimination of these restrictions in favor of increasingly "realistic" environments.

### 3.1.3 Boundary Function

OR assumes a strict boundary between the "inside" and the "outside" of a logical machine. Events on the inside are presumed unobservable and are allowed to be based on uncommitted (or even incorrect) computations which may be rolled back. Events on the outside are presumed observable, may not be rolled back, but may be delayed until they are committable.

Decisions on where to draw the boundary affect performance, ease of implementation, and the extent of the recovery. For example, ideally the window manager is *inside* the logical machine. Then if the machine is shut down or the power fails, on restart all windows will appear just where they were "recently" before the failure. In the current prototype, we chose not to modify the Mach kernel, but instead to recompile and relink recoverable programs with a modified Mach library and *mach.h*. It was not feasible at the time to remake the window manager, and therefore the windows are currently *outside* the logical machine. In our test system, we make an arbitrary assumption that each RU owns a single *xterm* window. The window is rebuilt when the RU recovers. In a future version of our prototype, we expect to locate the window manager *inside* the logical machine.

For performance reasons we must relax the strict output boundary function. If every key stroke and mouse



motion is an input event, and every echo of a key stroke and every position change of the mouse cursor is an output event, there would be a need to delay the echo of a key until its causal antecedent (the keystroke) was logged, which is impractical, even with OR. Even in failure-free environments, the end user is accustomed to seeing "uncommitted" states of a screen—for example, intermediate bit patterns while windows are being moved around—and only trusts what is on the screen once it is "stabilized". We therefore decided that screen output was not considered to have been "observed" unless it was on the screen for  $k$  milliseconds without being undone by a failure or recovery action.

Another component where there is a choice of where to draw the boundary is the file system. The file system could be viewed as an output device, receiving only committed data, but it is also possible to view the file system as the internal state (volatile storage) of an RU corresponding to the file server. There are optimizations to avoid logging all messages to a file server[1], and to avoid copying the whole file system in order to take a checkpoint[20]. The current implementation does not support recovering the state of the UNIX file system.

### 3.1.4 Piecewise Determinism

OR presupposes that each RU is piecewise deterministic. For a single-threaded Mach task without shared memory, this is straightforward. The computation and most of the system calls are deterministic. In certain cases the Mach documentation is not clear on whether a system call has non-deterministic results—e.g. *vm\_allocate* for a nonspecific address. Those system calls with non-deterministic results must log their results, although if there is a usual result (e.g. *msg\_send* completing without timeout), it suffices to log only the unusual results.

For a multithreaded Mach task, there can be nondeterminism unless the order of access to shared resources can be made deterministic, or it can be logged, or some determiner of this order can be logged.

Multithreading can be made approximately deterministic using the C-threads coroutine package, since there is actually only one thread which includes a C-threads scheduler. However, this implementation uses timeouts, and it would be necessary to log not only input messages, but timeouts as well.

If multiple Mach threads are run in parallel, then they could access shared memory and other shared resources such as ports in a non-deterministic order. Since our recovery layer intercepts system calls to operate on ports, it would be easy to log the order of these operations. However, it is difficult to trap the change of "ownership" of virtual memory without getting into the kernel and obtaining the assistance of the memory mapping hardware. We view this as impractical in the absence of further knowledge of the application structure such as knowing that the application was written in a high-level language with particular disciplined use of memory.

However, for an important disciplined use of shared memory (the concurrent reader, exclusive writer assumption), we *can* make execution piecewise deterministic provided that we log access to operations such as mutual exclusion (mutex) which the application needs to issue to guarantee correct usage. On replay we will force threads to obtain mutex locks in the order shown in the log, by blocking out-of-order operations. This guarantees that the shared resources will be accessed in the same order. Since Mach does not guarantee any particular coherence property to shared memory, (e.g. atomicity), it is unlikely that any program making undisciplined use of shared memory will be portable. Hence we expect that most applications will access shared memory through packages like C-threads which enforce some discipline such as concurrent reader exclusive writer. In particular, the Ada language requires its shared memory applications to obey these assumptions.

Our implementation is piecewise deterministic provided these assumptions about shared memory utilization within a task are met. How severe a restriction this is, how violations can be detected, and whether this restriction can be relaxed is a subject for further study.

## 3.2 Implementation Problems

Our work is in a preliminary stage, so we can offer only tentative conclusions about the problems of implementing transparent optimistic recovery on Mach.

On the one hand, the Mach philosophy of supporting a thin kernel with minimal state, with most system services implemented on top of Mach, and with a "small" set of primitives based on message passing is a good fit for the assumptions of optimistic recovery. In fact most of our problems are side effects of Mach's not having gone far enough in exploiting the Mach philosophy.

Current problem areas are:

- Undefined semantics: The current Mach documentation is not always mathematically rigorous. Often we have to determine the semantics experimentally, and there is always the risk that we have interpreted a bug or release dependent feature as part of the definition. In particular, it is not clear what Mach promises or does not promise to be true of non-failing systems which are connected to failing systems.
- State in the kernel: The ideal operating system for OR is one with a completely stateless kernel. Any kernel state other than state used purely for performance enhancement needs to be made recoverable and needs to be "assigned" to a single RU.
- Source vs. Object Compatibility: We currently need to recompile Mach programs with a different include library and link them with a different run-time library. If we were able to intercept Mach kernel calls, we would be able to be Mach-compatible at the object code level. Unfortunately, because application programs actually build Mach message headers, we would introduce some inefficiency in replacing these with our different and slightly larger headers.
- Duplication: We have chosen to build our prototype on top of Mach. We would make kernel modifications only as a last resort. Unfortunately this often causes us to duplicate some work which Mach is also doing transparently to us. For example, we need to track when receive rights change ownership, so that we can associate messages with the appropriate OR link. However Mach needs to do approximately the same work itself. Another example: OR "sessions" duplicate some of the protocol which Mach already provides to support reliable message delivery.
- Security: Because we have chosen not to modify the kernel, we are storing data structures needed for recovery in the virtual memory of the RU being recovered. A buggy program could modify out these data structures, cause OR to checkpoint or log bad data, and thereby prevent recovery.

## 4 ORM Prototype

ORM supports the following subset of the Mach interface:

- Ports: *port\_allocate*, *port\_deallocate*, *task\_notify*, *task\_self*, *environment\_port*
- Messages: *msg\_send*, *msg\_receive*, *msg\_rpc*
- Environment manager: *env\_set\_port*, *env\_get\_port*
- Miscellaneous: *scanf*, *printf*, *exit*

This subset is enough to write fairly complicated distributed applications.

A programmer writes code as though programming for a vanilla Mach system, with the following additions:

- The file *OR.h* must be included in each C file—this file contains a list of macros that replaces all occurrences of the calls given above by the corresponding OR calls. For example *port\_allocate* is replaced by *PortAllocate*, where the latter is a routine we provide. In addition, it includes the data structures and globals that the user program must use. In particular, it defines an expanded message header structure, *MsgHeader\_t*.
- A call to *ORInit* must be made in *main* before anything else is done; this call is discussed below. In the future we can avoid this by having the standard startup code make the call.

## 4.1 Overview of ORM

Suppose a programmer writes a multi-task Mach application, and wishes to make it recoverable. How does the programmer use our OR layer?

Each RU consists of two parts—user code, and OR code. Although the user sees no difference in the functioning of the program (save that the program becomes recoverable), several things are actually going on. First, OR creates several Mach threads in the task. Each of these threads is responsible for a particular piece of work—for example, asynchronous logging of messages, asynchronous checkpointing, updating log maps etc. Furthermore, all user calls to the supported subset of the Mach interface are replaced by calls to equivalent OR procedures.

Before running the application, the programmer must ensure that two special tasks, the *Recovery Manager* (RM) and the *Port Authority* are running.



## 4.2 Reliable Ports

Informally, a *reliable port* is a port that survives task and communications failures.

For example, suppose a Mach task holding receive rights to a Mach port *p* fails. Recovering the task is not sufficient to mask the failure, since other tasks holding send rights to the port can no longer send to the task.

However, when a failing task *t* represents an RU *r*, the above behavior is undesirable. We want recovery of *t* to replace it by another task *t'*, and for *t'* to acquire all the port rights that *t* owned. Furthermore, the acquisition must be transparent to the user code in any RU.

The reliable port abstraction is our solution to the above problem. A Mach application using OR manipulates only *reliable* ports, instead of Mach ports. Using OR replaces the type of a Mach port with the type of a reliable port so all user code that handles Mach ports actually handles reliable ports. The OR code maps reliable ports into Mach ports and manages the Mach ports required for actual communication.

Our design is based on the use of an additional name-server, called the *Port Authority* (PA). The PA maintains a table of mappings from reliable port to Mach port. In addition, it maintains a *dependency map* associated with each entry.

The operations we support include the creation of reliable ports, the deletion of reliable ports, the passing of send rights and the passing of receive rights. Furthermore, we ensure that all the information stored in the PA is treated only as a *hint*. There are two major advantages to this. First, we avoid any data consistency problems that might arise when the PA is replicated across different machines. Second, there is no need to reliably log any of the information in the PA.

In this section we will deal with that part of the design that has actually been implemented in ORM including reliable port allocation and deallocation, passing send rights and recovery from failure. In Section 4.2.2 we will talk about the aspects of the design that are not present in ORM; primarily the passing of receive rights.

### 4.2.1 The Port Authority

In principle, a recovering RU can bind its reliable port rights to the Mach ports held by other RUs by inter-

acting directly with the other RUs. The PA serves as an intermediary for these interactions, caching bindings from reliable port identifiers to Mach ports. Therefore, the PA is simply a convenience for the implementation of reliable ports, and need not be made reliable itself.

The ORM Port Authority interface supports three operations:

- **void**  
PA\_Advertise(*r\_port*, *m\_port*)  
ReliablePort\_t *r\_port*;  
port\_t *m\_port*;  
An RU makes this call to bind the reliable port to the Mach port. It passes the send rights for the Mach port *m\_port* to the PA.
- **port\_t**  
PA\_Lookup(*r\_port*)  
ReliablePort\_t *r\_port*;  
An RU makes this call to obtain send rights to the reliable port, *r\_port*. The PA passes send rights for the Mach port it currently believes is bound to *r\_port* back to the RU which makes the call.
- **void**  
PA\_Free(*r\_port*)  
ReliablePort\_t *r\_port*;  
An RU makes this call to cancel any binding the PA has for the reliable port *r\_port*.

Although the PA is stateless, for ORM to function, the PA must always be present.

### 4.2.2 Receive Rights Transfer

Recall that ORM does not support the transfer of receive rights. In order to do so, we require the following additions:

- The PA must associate a dependency map with each of its bindings. The same port may be owned by RU *A* for part of its lifetime and by RU *B* for another part of its lifetime. The PA will try to cache the latest ownership.
- It may be necessary for senders to retransmit missing messages. It is important not to retransmit messages which have in fact been successfully received by a previous owner. Therefore each RU needs to track the session sequence number of the last received message prior to acquiring the port.

- OR in an RU that owns the receive rights to a port must keep track of all the active senders to that port. This information must be passed when the receive rights are passed.

### 4.3 *ORInit*

*ORInit* performs initialization for a new RU. It performs several functions:

- Assigns the RU a unique identification.
- Creates reliable ports and bindings for *task\_self*, *task\_notify*, *environment\_port*. These bindings are saved locally. They do not need to be registered with the PA.
- Initializes the other subsystems—half sessions, checkpointing, logging, interfaces with the PA and the RM.
- Creates the following threads:

**ORMain** Respond to OR messages and requests

**Log Daemon** Log messages asynchronously

**Retransmission Daemon** Retransmit unacknowledged messages

**Checkpoint Daemon** Checkpoint asynchronously

**Reinitialization Daemon** Restore a task's state from disk on recovery

#### 4.3.1 *task\_self*, *task\_notify*, *environment\_port*

All these calls return the reliable port bound to the appropriate port.

#### 4.3.2 *port\_allocate*, *port\_deallocate*

```
kern_return_t
port_allocate(r_task, r_port)
Reliable_Task_t r_task;
Reliable_Port_t *r_port;

kern_return_t
port_deallocate(r_port)
Reliable_Port_t r_port;
```

In response to a *port\_allocate*, OR first assigns a globally unique name *rp*. It then allocates a Mach port *mp*, and creates a local binding between *mp* and *rp*. Next, it registers the binding between *mp* and *rp* with the PA, via the call *PA\_Advertise*. Finally, it returns *rp* as the reliable port *r\_port* to be used by the application.

In **ORM**, a task can only allocate a port in itself, and not in another Mach task.

To reclaim resources on a *port\_deallocate* of reliable port *rp*, OR removes the local binding, gets rid of the Mach port bound to *rp* and asks the PA to free *rp*.

#### 4.3.3 *MsgHeader\_t*

We have added several fields to the Mach message header structure *msg\_header\_t*:

- The session sequence number for the half session protocol
- The unique identifier of a local reliable port
- The unique identifier of a remote reliable port
- The unique name of the sending RU
- A dependency map

Application code compiled with *OR.h* will automatically use our expanded message header, and not the original Mach message header. OR also renames the *msg\_local\_port* and *msg\_remote\_port* fields so the programmer is hidden from the substitution of Mach ports by reliable ports, and also there is no need for reformatting of the header by OR.

#### 4.3.4 Message passing

We support message passing via *msg\_send*, *msg\_rpc* and *msg\_receive*. Furthermore, we support the transfer of send rights to a reliable port via a message header.

Suppose that a task wants to send some data to another task, listening to some port, *remote*. Suppose also that the task wants to give the other task send rights to a port *local*. In the Mach paradigm, the task initializes a message header—in particular, the fields *msg\_local\_port* to *local* and *msg\_remote\_port* to *remote*—and performs either a *msg\_send* or a *msg\_rpc*.

Now consider **ORM**, and recall the we have modified the definition of a message header. The user proceeds as before—initializing the message header and making a call to the message send mechanism. However, the call is intercepted by **OR**, which first performs a translation based on locally cached information, of the user supplied reliable ports into Mach ports—say the reliable port named *remote* into *mach\_remote* and *local* into *mach\_local*. Then, additional information, such as a sequence number, the RU name, and a dependency map are added to the header. Finally, the message is sent using a Mach call.

Suppose that the Mach call fails because port *mach\_remote* is no longer valid. There are two reasons—either the port has been deallocated by the remote RU, or the remote RU has failed, and is being recovered. These cases can be distinguished by checking with other RUs. If no other RU is receiving on the port then it has been deallocated. Otherwise, eventually some RU will advertise the port with the PA.

When receiving a message, a Mach task must fill in a message header, including the port on which the message is to be received. This is followed by a call the *msg\_receive*. When the task is made recoverable, the *msg\_receive* call is intercepted by **OR**, which translated the reliable port inserted by the user, into a Mach port.

Note that timeouts are possible in both message sends and receives. To permit deterministic playback after a failure, this timeout information is logged.

#### 4.3.5 Summary of reliable ports in ORM

Any reliable port has a globally unique name. When a reliable port is allocated, the PA is given send rights to a newly allocated Mach port that is bound to that reliable port. Send rights to this Mach port are passed whenever send rights to the reliable port are passed. When an RU obtains rights to a reliable port, it caches the rights to the associated Mach port.

When an RU fails and recovers, it allocates a new Mach port for each reliable port for which it has receive rights. This new binding is sent to the PA (like when the reliable port was first allocated).

When an RU deallocates a receive reliable port, it informs the PA to delete any binding for that reliable port.

When an RU wants to send a message along a reliable port, it uses the locally cached Mach port (bound to that reliable port). If this port is bad, the RU continues to perform lookups on the reliable port in the PA, until one of the following:

- either the PA is able to return a good Mach port—corresponding to the new port installed by a recovered receiver, or
- the PA is unable to find a binding for that reliable port—corresponding to a deallocation of the port.

Thus, the overhead incurred in the absence of a failure is the registration of a reliable port with the PA. Subsequent message exchanges require only a local translation between reliable port to (a cached) Mach port. In the case of failure however, the receiver will have to register the new Mach port, and each sender will have to obtain send rights to that new port from the PA.

#### 4.3.6 The ORM Recovery Manager

The **ORM Recovery Manager** is a task that checks to see if an RU has failed. If so, it tries to restart the RU from the appropriate checkpoint.

Note that the **RM** is stateless, as all the information required about running tasks is already available in the file system. Thus, there is no need to ensure that the **RM** checkpoints itself. **RM** is needed for automatic failure detection. If **RM** is absent, failure detection and RU restart must be performed manually.

#### 4.3.7 Information logged

In **ORM**, the following information is asynchronously logged:

- Messages received—this causes an update of the log map. Additionally, an acknowledgement is sent to the RU that originated the message to prevent retransmissions.
- Inputs (via *scanfs*)—this causes an update of the log map.
- Timeouts on message sends/receives—this is required for deterministic playback on failure.

## 4.4 Saving and Restoring Checkpoints

Implementing rollback requires the preservation of the full state of a task. Our initial implementation checkpoints a task state by forking a child whose state is written out by an OR thread to disk while the parent continues execution.

In order to recover from a failure, the OR mechanism must take checkpoints of the RU states periodically. The states should include

- Executing states of all threads.
- The virtual memory image of the task.
- Mach ports
- Other Unix-related state: signal, file system, pipe, ...etc.

The first two are necessary to restore a failed task. They can be retrieved through Mach kernel calls. We do not save the kernel state associated with Mach ports. Instead, the reliable port abstraction that we have done will handle the re-creation of Mach ports transparently. We save the name of a reliable port if the port for which this task has receive rights. When the task is restored, this information is used to generate a new Mach port and to advertise it to the PA. For those reliable ports to which the task has send rights, the recreation of Mach ports is not done until the restored task attempts to send messages on these reliable ports. At that time the PA can supply the necessary Mach ports and rebind them with reliable ports. For the Unix states, they are maintained by the kernel and there is no general way to extract them from the kernel, other than intercepting all Unix calls, recording its state, and forwarding the call to the Mach kernel. Currently we do not save these Unix states.

In our implementation of ORM, each task has several OR daemon threads and user threads. In particular, the Checkpoint Daemon checkpoints its own task and saves necessary states, including thread states and virtual memory states, into disk by the following algorithm, the necessary Mach kernel calls are enclosed in parenthesis:

- Suspend all user threads and other daemon threads (`thread_suspend`)
- Allocate a thread buffer to store all thread states (`task_threads`, `vm_allocate`)

- For each threads, abort the thread (`thread_abort`), and get its states into the buffer (`thread_get_state`)
- Do a *unix\_fork*

The *unix\_fork* copies the current task into a child task with a single thread. The child task serves as a virtual memory image of this task, while the Checkpoint Daemon in the parent task takes the checkpoint. The reason to use a *unix\_fork* instead of a *fork* is to get the virtual memory image of *all* threads; a Mach *fork* deallocates the virtual memory of all but the calling thread.

After a *unix\_fork*, the child task suspends itself. The Checkpoint Daemon thread will

- Resume all user threads and other daemon threads (`thread_resume`)
- Write each virtual memory region of the child task into disk (`vm_region`, `vm_read`)
- Write out the thread state buffer and deallocate the buffer (`vm_deallocate`)
- Close the checkpoint file and terminate the child task

Instead of resuming all other threads after the checkpoint is written out, OR resumes them immediately after the *unix\_fork* to minimize the freezing time. Note the Checkpoint Daemon can not checkpoint itself. Thus it must be recreated when OR restores a task.

Note that the current implementation even checkpoints OR daemon threads, which is not necessary since they can be recreated after a restore.

To restore a task is rather straightforward. First the restorer does a *unix\_fork* to create a clone task which serves as a template for the new restored task. The clone task simply suspends itself. Then the parent task performs the following actions in the context of the clone task:

- Terminate all threads (`thread_terminate`)
- Deallocate each memory region (`vm_region`, `vm_deallocate`)
- Create and load memory regions of the restored task (`vm_allocate`, `vm_write`, `vm_protection`)



- Create each thread (`thread_create`) and set their state (`thread_set_state`)
- Resume the reinitialization thread—Reinitialization Daemon (`thread_resume`)

The reinitialization thread reinitializes all OR and user related states, creates the Checkpoint Daemon, and resumes all other threads.

## 5 Dependency Tracking

Dependencies are stored as a *dependency map* (DM) which is a table mapping RU  $i$  to incarnation, interval pair, expressed as  $[\iota, \mu]$ , where  $\iota$  is the incarnation and  $\mu$  the sequence number of the latest message received by RU  $i$  that causally proceeds this state. Each user task has a current DM (Task\_DM) indicating the latest interval of each task that it depends on. The optimistic computation tracks dependencies by inheriting them in states and messages. When application code sends a message, OR transmits a copy of Task\_DM with the message. Similarly, when an RU takes a checkpoint or application code outputs external data through the output boundary function, OR copies the Task\_DM into the checkpoint and the output message.<sup>3</sup>

When an RU receives a message either from another RU or from the external input boundary function, OR increments the Interval number for the RU's entry in its Task\_DM. When an RU receives a message from another RU, OR merges the message's DM into the RU's Task\_DM, setting the entry in Task\_DM to the most recent antecedent of the DM in the message and its Task\_DM. If both maps contain RU  $X$ , one as  $(\iota_1, \mu_1)$  and the other as  $(\iota_2, \mu_2)$ , then the most recent antecedent is given by

- $(\iota_1, \mu_1)$  if  $\iota_1 > \iota_2$
- $(\iota_2, \mu_2)$  if  $\iota_1 < \iota_2$
- $(\iota_1, \max(\mu_1, \mu_2))$  if  $\iota_1 = \iota_2$

Absence of an RU entry in a DM means that the computation at that point does not depend on the RU. Ex-

<sup>3</sup>We currently store the DM in an enlarged header, `Msg-Header.t`, in an in-line Mach message. In the future we plan to support out-of-line messages, and store the DM in an out-of-line buffer. Reducing the number of message formats from 3 to 2 would simplify this work.

ternal addition of another RU to an OR system simply adds another entry to the DMs that depend on it.

### 5.1 Incarnation Start Table

The Incarnation Start Table (IST) is a function from an RU, incarnation pair to interval.  $IST(R, \iota) = \mu$  means that RU  $R$ 's first action in incarnation  $\iota$  was to receive the message that started state interval  $\mu$ . By convention,  $IST(R, 1) = 0$  since an RU can send messages when it starts up.

The IST is used to identify orphan messages. A message is an orphan if it depends on a discarded, or orphan, message as indicated by the IST. For example, if  $IST(X, 2) = 4$  and  $IST(X, 3) = 10$  then a message depending on state  $(2, \iota)$  of RU  $X$  with  $i \geq 10$  is an orphan. In general, an object depending on state  $(\iota, \mu)$  of RU  $X$  is an orphan if there exists  $IST(X, \iota)$  such that  $i > \iota$  and  $IST(X, \iota) \leq \mu$ . Note that this is a one way implication—a message that is not now an orphan can become an orphan in the future.

A running RU always knows its own IST. The RU stores its IST on stable storage.

### 5.2 Recovery

An RU starts a new incarnation after each rollback. On a rollback in incarnation  $\iota$  RU  $X$ :

1. restores its oldest checkpoint
2. re-receives input messages from its log, stopping before the first orphan
3. if a failure occurred  $X$  reads the IST from stable storage
4. writes  $IST(X, \iota+1) = \text{current receive sequence number} + 1$  to stable storage
5. sends `RECOVER`( $X, \iota+1, \text{current receive sequence number} + 1$ ) to all other RUs
6. re-receive the remaining non-orphan messages from the log
7. begins communicating with other RUs

An RU also knows the IST function for other RUs although this may be out of date and incomplete. The



IST is communicated in RECOVER messages, sent during recovery (above) and on request from an RU missing IST data.

### 5.3 Rollback

Since the IST determines whether a message is an orphan only two events can cause a rollback:

1. a failure
2. receipt of a RECOVER message that updates the IST

The IST for  $X$  at RU  $Y$  can be incomplete. A single failure of  $X$  should not cause this since  $Y$  will receive  $X$ 's RECOVER message, but a multiple failure may lose RECOVER messages. RU  $Y$  detects an incomplete IST when it receives a message that depends on incarnation  $\iota$  at  $X$  and finds that  $IST(X, \iota) = \text{UNDEFINED}$  for some  $i \leq \iota$ .  $Y$  must request a RECOVER( $X, \iota$ ) message from  $X$  for all values of  $i \leq \iota$ .

A message or state with a particular dependency map becomes "committable" when all the messages it depends on have been logged. We assume an RU logs messages in receive sequence number order. A log map maps each element of a set of RUs to an incarnation, interval pair indicating the extent of the RU's logging. When an RU logs a set of input messages it updates its local log map and broadcasts the map to other RUs. Note that a logged message can later become an orphan.

To determine whether a dependency map is committable we compare it with the local log map. If RU  $X$  is only in the dependency map then it is not committable; if it is only in the log map then ignore it. If RU  $X$  is in the dependency map as  $(\iota_D, \mu_D)$  and the log map as  $(\iota_L, \mu_L)$ , then the dependency map is not committable if  $(\iota_D > \iota_L)$  or  $(\iota_D = \iota_L \text{ and } \mu_D > \mu_L)$ . (We assume that the IST for  $X$  is complete up through  $\iota_D$  so that orphans have been already eliminated.) Otherwise it is committable.

An RU cannot execute a message that depends on an interval whose IST is undefined. When it discovers the IST is incomplete, it delays execution of the message until the requested RECOVER message is received. When a RECOVER message arrives, the RU compares its dependency map with the IST update. If the dependency map depends on an orphan then the RU rolls

back. In this way an RU effects any rollbacks for incarnation  $C$  before it starts depending on incarnations greater than  $C$ .

### 5.4 Output Boundary Function

OR produces output as soon as possible after it is committed. The Output Boundary Function (OBF) buffers output until it has been committed.

The current prototype replaces the *printf* with a function that converts the output into a string and passes the string to the OBF. Whenever an RU's log map is updated OBF attempts to output results that have been newly committed. In the future we plan to support the full stdio package by relinking the write calls stdio makes to a write function that buffers output until it is committed.

#### 5.4.1 Stuttering

OR can stutter because logging a message that commits output and producing the output must be separate events. To minimize stuttering the prototype stably stores a count of the messages output by the OBF, which we update after completing the output. During the replay after recovery we discard all output with sequence numbers less than or equal to the stably stored count. Stuttering is still possible, since a crash between some output and incrementing the stable count will reproduce the output during the replay after recovery, but it is unlikely.

### 5.5 Logging Messages and Non-deterministic Events

A key challenge in implementing optimistic recovery, or indeed any system relying on rollback and replay, is logging all events that determine a task's behavior. The most obvious determinant of a task's behavior are the messages it receives. A message can be viewed as two parts: its content, and its position in the receive sequence at the task. The content can be logged either by the receiving task, as in ORM, or by the sending task, as discussed in [17, 10]. The merge position depends on unpredictable message communication delays, so must be determined at the receiver and should probably be logged there.

Another source of non-determinism in Mach is memory sharing between tasks, or between concurrent threads within an address space. On a uniprocessor, a deterministic thread scheduler (such as coroutine threads [3]) can be used to make the execution deterministic. We do not have a way of addressing this problem on a true multiprocessor, so ORM will initially be restricted to tasks running on a single cpu and without shared memory between tasks.

There is a third source of non-determinism in Mach which we had not expected, namely the ability of one task to modify the state of another task through a system call [15]. In some cases, such as the de-allocation of a port by another task, the effect is only seen when the former owner of the port tries to perform an operation on the port – in this case, logging of the non-deterministic event can be deferred until its effects become visible for the first time. In other cases, such as one task mapping a page into the memory of another task, there is no system call to use as a handle for logging. This is similar to the shared memory problem, and will not be supported in the first version of ORM.

## 6 Comparison with other systems

### 6.1 Transaction Systems

The original recoverable systems were database systems, and it remains true that most work on recovery assumes a transaction model. For many, the very word “recovery” carries the implication that the user has specified an atomic unit of work. Many experimental recoverable systems, such as Argus[12], Quicksilver[7], and Avalon/C++[8] assume a transaction model.

In transaction models the transactions or units of work must be explicitly defined by the programmer. Transactions run in parallel and share the database; typically the “recovery” mechanism enforces both atomicity and recovery.

The major differences in functionality between OR and transaction systems are: (1) transaction systems recover only some of the data, e.g. the database but not the program counter and local variables of long-running actions; (2) programmers must explicitly define units of work, and hence a conventional program cannot be transparently made recoverable; (3) at the end of a dis-

tributed transaction a two-phase commit is initiated and at least one record must be force-written, whereas OR continuously logs information to stable storage and continuously commits computations whose dependencies are all logged.

### 6.2 ISIS

ISIS [11] is a distributed programming environment that has fault-tolerance as one of its goals. It provides the user with a variety of tools, including ones that allow the replication of a user-defined part of the process state, the maintenance of process groups, and so on. Using these tools, a programmer can create fault-tolerant applications. However, an efficient use of the ISIS tools calls for sophisticated knowledge and system design ability, and may be beyond most programmers.

In contrast to ISIS, in Optimistic Recovery, the programmer can be oblivious to the possibility of failure, and can write an application for the Mach interface, as though failures do not occur. This represents a considerable simplification over having to explicitly code in fault-tolerance.

## 7 Future Plans

One can classify our future plans into *realism* enhancements and *performance* enhancements. The realism enhancements will enable the OR layer to support a larger set of Mach and Unix primitives. This work includes:

- Link the stdio package with reads that call the input boundary function and writes that call the output boundary function.
- Support communication via out-of-line Mach messages.
- Support passing receive rights to reliable ports.
- Support dynamic creation of RUs (fork() or task\_create()) and destruction of RUs (exit()). These operations must be made undoable, so that rollback can ‘un’create or ‘un’destroy an RU.
- Implement the Mach port set operations and the virtual memory operations.
- Recover the state of a task’s user interface, by incorporating the state of the window system, such as X, in the RU.

- Make an object-level compatible Mach/OR system which will automatically recover applications .
- Support more Unix functionality, so that signals, pipes and file connections are recoverable. This work's architecture will depend on the implementation of the Unix outside the mach kernel.

Before making performance optimizations, we plan to measure the performance of **ORM** and tune it appropriately. Many performance optimizations will be based on the work in the earlier papers, [17, 1, 19, 20].

- Garbage collect checkpoints and message logs
- Experiment with different logging mechanisms, such as sender-based logging, volatile logging, and null logging.
- Optimize file system access.

We also intend to use the general rollback capability of **ORM** to examine other optimistic systems, such as Time Warp ([9]), optimistic process replication ([6]) and optimistic parallelization ([2]). Other applications for generalized rollback are also inviting, such as distributed debugging.

## 8 Conclusion

We have built a prototype software layer on top of Mach that recovers failed multi-task Mach applications. The layer implements Optimistic Recovery (OR), which asynchronously logs inter-task messages to disk, and occasionally checkpoints each Mach task's state. OR recovers from failure by restoring a checkpoint and replaying the logged messages. Dependencies are transmitted with each Mach message and logging progress is broadcast. OR prepares output as early as possible, but delays writing to the screen until the messages the output depends on have been logged.

Our extensions to Mach for rollback and replay have other applications besides recovery. For example, rollback, or the ability to checkpoint a task and continue its execution in a different operating environment, effectively migrates a process from a machine, to a disk and then back to a machine, whereas normal process migration skips the disk. Another application that uses rollback are distributed debuggers of non-deterministic programs [4, 13, 5].

We hope to convince the Mach community of the need for services that allow straightforward checkpointing, since we believe it is a critical service for fault-tolerance, migration, and debugging. In particular, state in the kernel must be accessible in a form that can be used independent of the running kernel. We have addressed a major portion of this problem in Mach with our reliable port mechanism. A similar degree of effort will be needed to provide reliable pipes, reliable sockets, etc.

We hope to create an awareness that sources of non-determinism should be limited, and where feasible implemented in a way that they can easily be recorded. For programs that communicate only by Mach message, we have demonstrated that this can be done.

## References

- [1] BACON, D. F. How to log all filesystem operations (while only writing a few to disk). Research Note RC, IBM T.J. Watson Research Center, 1990.
- [2] BACON, D. F., AND STROM, R. E. Optimistic parallelization of communicating sequential processes. Research Note RC, IBM T.J. Watson Research Center, 1990.
- [3] COOPER, E. C., AND DRAVES, R. P. C threads. Tech. rep., CS Department, CMU, October 1988.
- [4] FELDMAN, S. I., AND BROWN, C. B. IGOR: A system for program debugging via reversible execution. In *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* (May 1988), pp. 112-123.
- [5] FORIN, A. Debugging of heterogeneous parallel systems. In *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* (May 1988), pp. 130-140.
- [6] GOLDBERG, A. P., AND JEFFERSON, D. Transparent process cloning: A tool for load management of distributed systems. In *Proceedings of 1987 International Conference on Parallel Processing* (August 1987), pp. 728 - 734.
- [7] HASKIN, R., MALACHI, Y., SAWDON, W., AND CHAN, G. Recovery management in quicksilver. *Transactions on Computer Systems* 6, 1 (1988).
- [8] HERLIHY, M. P., AND WING, J. M. Avalon: Language support for reliable distributed systems. Tech. Rep. CMU-CS-86-164, Carnegie Mellon University, 1986.

- [9] JEFFERSON, D. Virtual time. *ACM Transactions on Programming Languages and Systems* 7, 3 (July 1985), 404.
- [10] JOHNSON, D. B., AND ZWAENEPOEL, W. Sender-based message logging. In *The Seventeenth International Symposium on Fault-Tolerant Computing* (June 1987), IEEE Computer Society, pp. 14-19.
- [11] KENNETH P. BIRMAN, R. C., ET AL. The isis system manual, version 2.0. Tech. rep., CS Department, Cornell, March 1990.
- [12] LISKOV, B., CURTIS, D., JOHNSON, P., AND SCHEIFLER, R. Implementation of argus. In *The Eleventh Symposium on Operating Systems Principles* (1987), ACM Special Interest Group on Operating Systems.
- [13] PAN, D. Z., AND LINTON, M. A. Supporting reverse execution of parallel programs. In *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* (May 1988), pp. 124-129.
- [14] RANDELL, B. System structure for software fault tolerance. *IEEE Transactions on Software Engineering SE-1*, 2 (June 1975), 220-232.
- [15] ROBERT V. BARON, D. B., ET AL. Mach kernel interface manual. Tech. rep., CS Department, CMU, April 1990.
- [16] SCHNEIDER, F. B. Fail-stop processors. In *Digest of Papers from Spring Comcon* (March 1983), IEEE Computer Society.
- [17] STROM, R. E., BACON, D. F., AND YEMINI, S. A. Volatile logging in n-fault-tolerant distributed systems. In *The Eighteenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers* (June 1988), pp. 44-49.
- [18] STROM, R. E., AND YEMINI, S. A. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems* 3, 3 (August 1985), 204-226.
- [19] STROM, R. E., YEMINI, S. A., AND BACON, D. F. Toward self-recovering operating systems. In *The International Conference on Parallel Processing and applications* (L'Aquila, Italy, Sept. 1987), North-Holland, pp. 475-483.
- [20] STROM, R. E., YEMINI, S. A., AND BACON, D. F. A recoverable object store. In *Hawaii International Conference on System Sciences* (Kailua-Kona, HI, Jan. 1988), vol. II, pp. 215-221.
- [21] WALMER, L. R., AND THOMPSON, M. R. A programmer's guide to the mach system calls. Tech. rep., CS Department, CMU, December 1989.





# Fault-Tolerant Computing Based on Mach\*

Özalp Babaoğlu

Department of Mathematics  
University of Bologna  
Piazza Porta S. Donato 5  
40127 Bologna, Italy

2 August 1990

## Abstract

We consider the problem of providing automatic and transparent fault tolerance to arbitrary user computations based on the Mach operating system. Among the several alternatives for structuring such a system, we pursue the "task-pair backup" paradigm in detail and outline how it might be supported by Mach. Some of the new system calls and protocols that need to be incorporated into the Mach kernel and server tasks are sketched.

---

\*This work was supported in part by the United States Department of Defense Advanced Research Projects Agency (DARPA) under grant N00140-87C-8904, the Commission of the European Communities under the ESPRIT Programme Basic Research Action Number 3092 (Predictably Dependable Computing Systems) and the Italian Ministry of Research and University. The views, opinions, and findings contained in this report are those of the author and should not be construed as an official position, policy, or decision of the funding organizations.

# 1 Introduction

Mach [1] is an example of the kernel-based technology for constructing operating system environments on multiprocessor and distributed systems. Other systems that subscribe to the same philosophy for structuring computing environments include the V kernel [8], Chorus [17] and Amoeba [16]. Among these systems, Mach has emerged as a particularly popular foundation for UNIX<sup>TM</sup> development on a wide range of interesting architectures.

The properties that make Mach appropriate for distributed computing (e.g., message-based interactions, many services being performed by user-level tasks, etc.) would appear to make it appropriate for fault-tolerant computing as well. In this work, we pursue this possibility and propose an architecture to support fault-tolerant user computations based on Mach.

The following section outlines the requirements for the hardware environment as it relates to fault tolerance. Section 3 gives an overview of Mach and considers the problem of structuring it on the hardware base. In Section 4 we identify the goals for fault tolerance that are desirable and feasible in the context of Mach. In light of these goals, various architectures for fault tolerance are discussed in Section 5. The paradigm of backing up each task with an inactive secondary is explored in detail in Section 6. We conclude the paper by listing problems that remain to be solved before the architecture can be realized.

## 2 The Hardware Environment

We consider a generic multiprocessor system as the hardware base. The system consists of multiple autonomous processing units called *nodes* that may be tightly coupled through a parallel bus or loosely coupled through a network. There may or may not be support for shared memory at the hardware level.

We make the following assumptions about the failure characteristics of the hardware:

1. Processors fail by stopping. In other words, a fault processor never makes incorrect state transitions that are externally visible.
2. The hardware exhibits failure independence in the sense that the failure of one node does not affect the operation of other nodes.
3. There are no critical components whose failure would render large parts of the system inoperative. Examples of such components include power supplies, interconnection structures and communication links. If present, we assume that these components are either sufficiently reliable or are replicated a sufficient number of times so as not to be considered critical.
4. The communication subsystem may lose messages. Message delivery times are bounded and can be used to detect processor failures. Processor and communication failures never partition the system.

A large number of commercial multiprocessor systems have been constructed that satisfy the above assumptions to varying degrees. In particular, a conventional geographically distributed, loosely-coupled network architecture comes very close to meeting all of the assumptions. In such a system, the communication channel (e.g., Ethernet) is typically the only critical component and it could be easily replicated to achieve the desired failure modes of the hardware.

## 3 The Software

### 3.1 Mach Overview

Mach is a communication-oriented operating system that is particularly well suited for multiprocessor and distributed architectures. It consists of a small, extensible kernel and a collection of user-level tasks that provide (perhaps several) operating system support environments. The abstractions supported by the Mach kernel are the following:

**Tasks** are the basic units of resource allocation and define an environment for execution.

**Threads** are the basic units of execution. Threads within the same task share its resources (including virtual memory). A thread contains the minimal state information to guarantee independent execution.

**Ports** are simplex communication channels that are protected by the kernel using capabilities. All references to objects in Mach are performed by sending messages to ports representing them.

**Messages** are typed collection of data that are communicated between threads.

**Paging Objects** are secondary storage objects associated with a region of a task's virtual memory.

The Mach kernel itself can be considered a multi-threaded task that accepts messages corresponding to the kernel operations from user tasks. The procedural system call interfaces of the Mach kernel and servers are written in a remote procedure call language called MIG.

In addition to the primitives supported directly by the kernel, Mach provides complete 4.3bsd UNIX emulation. Currently, much of this emulation is supported directly in the Mach kernel. The so-called "de-kernelization" effort of moving all of the 4.3bsd emulation out of the kernel and into server tasks is ongoing but far from complete. In anticipation of an eventual separation of 4.3bsd emulation from the Mach kernel, we consider only the questions regarding the fault tolerance of the abstractions supported by the Mach kernel—tasks, threads, ports, messages and paging objects. Issues related to the fault tolerance of abstractions supported by server tasks (network server, memory server, file server, 4.3bsd emulation servers, etc.) can be addressed only when these server tasks are well defined. Furthermore, having fault-tolerant kernel objects at their disposal, many servers should attain fault tolerance at little or no additional effort.

### 3.2 Mach on a Fault-Tolerant Multiprocessor

In deciding a rational strategy for structuring Mach on a multiprocessor, it is useful to make the following distinctions based on the memory access characteristics of the architecture:

**Uniform Memory Access (UMA)** machines are shared memory multiprocessors where there is no difference in memory access costs with respect to regions of the physical address space.

**Non-Uniform Memory Access (NUMA)** machines, on the other hand, also share a common physical memory, but memory access costs may be different for different physical addresses (e.g., local vs. remote memory).

**No Remote Memory Access (NORMA)** machines classify physical memory as "local" and "remote" with respect to each processor and do not permit direct access of remote memory. This property is typical of distributed systems where processors can only communicate via message exchange.

Regardless of the actual underlying multiprocessor hardware, the fault tolerance considerations discussed in Section 2 require us to view the system as a NORMA model for structuring Mach. In other words, any multiprocessor hardware has to be seen as a loosely-coupled collection of nodes with no shared resources. Given that there can be single failures that crash an entire node, there must be at least one independent Mach kernel per node. The same observation leads us to conclude that threads belonging to a single task should all be scheduled within the same node.

This distributed system view is similar to those of Tandem [2], Stratus [21] and Nixdorf [6] commercial fault-tolerant systems, which are all based on loosely coupled multiprocessor architectures. The Sequoia system [3], however, chooses to retain the NUMA architecture at the cost of having to develop a custom operating system of great complexity. Both the Stratus and Sequoia systems employ special hardware (processor pairs with hardware comparators) for hardware failure detection.

## 4 Goals for Fault-Tolerant Computing

With respect to fault tolerance, applications can be broadly classified as belonging to two domains:

- Process control
- Transaction processing.

The factor differentiating the two domains is the acceptable time for recovery after a failure. In the case of process control, real time constraints typically require immediate recovery. This, in turn, implies sufficient continual redundancy in the computation such

that failures can be masked as soon as they occur. Resulting system cost is high—increased computational resources must be dedicated to replicate a single task, leaving reduced system capacity for other applications. Transaction processing applications, on the other hand, typically have less severe recovery deadlines—all that is required is that the system eventually recover from a failure. Resulting system costs can be kept sufficiently low in the sense that most of the hardware resources available are utilized for useful computation. Obviously, there is a continuum of such system designs that trade off recovery time for system overhead.

The fault tolerance goals we have for the Mach system are of the eventual recovery type after any single node failure. Note that a single node failure admits multiple failures as long as all of them are in the same node. This goal is consistent with our desire not to have to modify the hardware in any way and to obtain a system with high productive processing power in the absence of failures. Another goal of our design is the automatic and transparent integration of fault tolerance into user tasks. In other words, unmodified user programs that are desired to be fault tolerant should be automatically backed up and recovered when failures occur. This is in contrast to systems such as Tandem and Tolerant [22] that require reprogramming or preprocessing user programs to render them fault tolerant.

Our goals could be summarized as follows: Automatic and transparent tolerance of any single node failure with reasonable recovery delays and small failure-free operation overheads. In this sense, our goals are very similar to those of the Nixdorf TARGON/32 system.

## 5 Models for Fault-Tolerant Computing

Within the context of eventual recovery, computations can be structured in several different ways to achieve fault tolerance. The various approaches differ essentially in the manner in which they guarantee global system state consistency when parts of the computation have to be recovered from past states due to failures. Here we outline some of the principal models that have been proposed.

### 5.1 Transactions

In the transaction model [11], computation is divided into units of work called *transactions*. The system guarantees three properties for transactions: atomicity, serializability and permanence. Atomicity is with respect to failures in the sense that the execution of a transaction must be all or nothing—failures should never leave intermediate states of the transaction computation visible to other transaction. Serializability, on the other hand, requires that the concurrent execution of a several transactions should be as if they executed (in some arbitrary order) one after the other. Permanence guarantees that the computation makes finite progress despite failures.

The basic transaction model has been extended to distributed data models and nested transactions [15]. By definition, transaction boundaries always define consistent system states from which a computation can recover from. The major drawback of the transaction model is that fault tolerance cannot be integrated transparently to applications. Programs



must explicitly use the transaction paradigm by announcing the beginning and end of transactions within programs at opportune points. Furthermore, while the serializability requirement of transaction-based recovery is appropriate for database applications, it can be overly restrictive for general distributed computations.

Modern systems that adopt the transaction model as the basis for fault tolerance in distributed computations include Arjuna [18], Argus [13] and Camelot [19]. Mach does not support the transaction abstraction at the kernel level. Efficient distributed transaction management systems can be built on top of Mach as a collection of server tasks. Camelot is one such system.

## 5.2 Checkpoint-Rollback

An arbitrary distributed computation could be made fault tolerant without having to structure it as a collection of transactions. All that is required is a mechanism whereby computations can be restarted from some past state in response to failures. To prevent having to restart computations always from the very beginning and thus guarantee forward progress, the state of the failure-free execution is periodically saved to stable storage. The saved past states are called *checkpoints*. The act of restoring a computation to a past state is called *rolling back* and the interval during which recovery is taking place *rolling forward*. The interval with which checkpoints are taken is a system tuning parameter and establishes the relative costs of the failure-free execution overhead and recovery delays.

In a system in which computations interact by exchanging messages or sharing data, rolling back a failed computation to an arbitrary checkpoint may result in an inconsistent global system state [7]. Intuitively, rollback should never result in a system state in which there are computations that appear to have received messages (read data values) that have not yet been sent (written). The manner in which global system state consistency is guaranteed results in two distinct strategies.

### 5.2.1 Optimistic Recovery

The general strategy is to design the failure-free execution of the system by “gambling” that failures will not occur. If indeed they don’t, all is fine. If, however, the system encounters a failure, it must have collected sufficient information along the way to roll back those computations necessary to establish a consistent system state for recovery. For example, the scheme proposed by Strom and Yemini [20] allows checkpointing and message logging to stable storage occur asynchronously with computations but maintains the causality information necessary to allow the recovery algorithm to construct the global system state to roll back to. In a variant of the scheme [23], messages are logged in the nonvolatile memory of the sender rather than the receiver, resulting in even further asynchrony of stable storage writes with respect to computation.

An assumption common to both [20] and [23] is that computations are deterministic—an initial state and a sequence of messages to be received uniquely establishes the final computation state and the sequence of messages it sends. At the cost of additional complexity,

Koo and Toueg present checkpointing and rollback algorithms that are suitable even when computations are not deterministic [12]. By storing two checkpoints per computation, their algorithms can also tolerate failures during checkpoint and rollback operations. An unfortunate consequence of optimistic strategies is that recovery time is difficult to bound since, in addition to the failed computation, an arbitrary number of others may need to roll back.

### 5.2.2 Pessimistic Recovery

A reasonable alternative to the above strategy is to structure the checkpointing mechanism in a manner such that recovery only involves the computations affected by the failure. This has the desirable consequence that recovery is both simple and more predictable in the delays it introduces to the system. The cost, obviously, is shifted from recovery to checkpointing.

To prevent arbitrary computations from having to be rolled back due to a failure, pessimistic schemes synchronize checkpointing with global interactions of computations. For example, if computations were to be checkpointed after each send operation, rolling back only the failed ones to their most recent checkpoint would always guarantee global state consistency. To avoid the substantial delays associated with checkpointing to a stable store, pessimistic schemes typically contain checkpoints in the context of a *backup* computation on another processor.

Notable systems that employ pessimistic recovery include Tandem [2] and Auragen [5] (while the Auragen approach was later adapted and further refined by Nixdorf for their TARGON/32 System [6], we continue to refer to it as the “Auragen approach”). Tandem uses process pairs to achieve fault-tolerant computations. Whenever the primary process engages in an activity that may induce global interaction (perhaps indirectly), it checkpoints its state to the backup. The responsibility for identifying these interactions and invoking the appropriate operating system primitives to do the checkpointing belongs to the programmer. The Auragen system, on the other hand, achieves application-independent fault tolerance automatically by performing checkpoints at fixed intervals but keeping the backup process always in a state where it can take over from the primary upon a failure. As with the schemes in [20] and [23], the Auragen method works only for deterministic computations. We elaborate on this scheme in the next section.

## 6 A Proposal for Fault-Tolerant Mach

In this section we outline a structure to render Mach tasks fault tolerant in a manner consistent with the goals of Section 4. The design we propose is essentially hardware independent—most distributed systems that have access to a fast broadcast communication medium could form the architectural foundation for the design.

Given the close harmony among our goals and the close match between the two computational models, we base our design on the Auragen approach. In this scheme, only user-level tasks can be made tolerant of single failures by backing them up on a processor with an independent failure mode than the primary. The operating system kernel is not backed up

in the sense that a crash of a node could cause the kernel state to be lost. Those services of the operating system that need to be fault-tolerant (because they contain state information that needs to survive crashes), have to be moved out of the kernel and into server tasks. This is exactly the philosophy adopted by Mach.

## 6.1 Overall Structure

Each node of the system executes an independent Mach kernel that manages the resources local to that node (thread scheduling on processors, physical memory, ports, etc.). The units of replication and, thus fault tolerance, are Mach tasks. Threads are implicitly replicated within the backup task. The basic idea is to selectively back up tasks on different nodes. The backup task remains inactive but is kept in a state not too far behind the primary by periodically checkpoints and presenting it all of the messages that are received by the primary. In this manner, should the primary fail, the backup becomes active and rolls forward by processing the messages enqueued for it and takes over as the primary when recovery is complete. Given the assumption that all tasks (more correctly, threads, since tasks do not execute) are deterministic, the final state reached by the recovered backup and the messages it sends are guaranteed to be identical to those of the primary. The scheme also includes a mechanism to suppress resending of messages already sent by the primary.

In the following sections we outline the extensions and changes necessary to the Mach kernel to realize this scheme.

## 6.2 Task and Thread Creation

The kernel primitive to create a new task is extended to be

```
task_create(parent_task, inherit_memory,  
            child_task, child_data, backup_type)
```

where the last parameter indicates how the child task is to be backed up. We discuss the various choices for backup type in Section 6.5. In case a backup is desired, it is created on a node different from the child (primary) task. In case the child is to inherit the parent's memory, it needs to be created on the same node as the parent task. The backup task is inactive in the sense that its threads are not selected for scheduling on the backup node. In Mach terminology, the task is created in the *suspended* state and as if a `task_wait()` call has been made. The kernel maintains a table that establishes the correspondence between the primary and the backup for the `child_task` and `child_data` ports. In other words, in trying to deliver a message to any port associated with the primary task, the kernel should have sufficient information to locate (perhaps with the help of a network server) the corresponding port at the backup.

Thread creation requires no changes as far as fault tolerance is concerned—the new thread is implicitly replicated within the backup task. The kernel, however, needs to record an association between the ports of the new thread and the backup task. This will be used

in making copies of future messages available to the backup thread that were sent to this thread.

### 6.3 Messages and Communication

In Mach, communication ports are associated with tasks. A message sent to a task port can be read by any thread within the task. Furthermore, ports can be declared to be **unrestricted** in which case a thread can choose to receive *any* message among those enqueued at the unrestricted ports. This possibility represents a potential violation of our assumption that all executions are deterministic. Two tasks may end up in different states even if they start in the same initial state and are presented the same message sequence if the two make different choices with respect to delivering messages to threads that have been received at unrestricted ports.

To remedy this problem and eliminate non-determinism, we propose that the

```
msg_receive(header, option, timeout)
```

kernel primitive be modified such that when invoked with the `msg_local_port` field of the message header set to `PORT_DEFAULT`, the first message from the port with the *smallest* identifier<sup>1</sup> is delivered to the thread. We also need to guarantee that port identifiers are assigned the same relative ordering at both the primary and the backup.

The major new mechanism that needs to be included in the kernel is a 3-way multicast facility. To keep the backups up-to-date with respect to primaries, a copy of each message sent from one task to another needs to be also delivered to the two backups (if they exist). In the most general case, a message sent from task *A* to task *B* needs to be delivered to the following three destinations:

- task *B* (the primary destination)
- task *B'* (the backup destination)
- task *A'* (the backup sender).

The situation is illustrated in Figure 1. The 3-way multicast needs to be atomic in that either all three or none of the destinations receive the message and that two concurrent broadcasts are received in the same (arbitrary) order at all destinations.

Note that in the case the message is destined to a control port associated with a thread, we treat it as if it were destined to the task containing the thread as far as the multicast is concerned. Another issue to be dealt with in the case of messages to thread ports is the situation where the corresponding port (and thus the thread) at the backup does not yet exist. In this case, the message to the backup has to contain sufficient information (for example the relative ordering of the port identifier discussed above) for the message to be

---

<sup>1</sup>Actually, any totally ordered field associated with the port is sufficient.



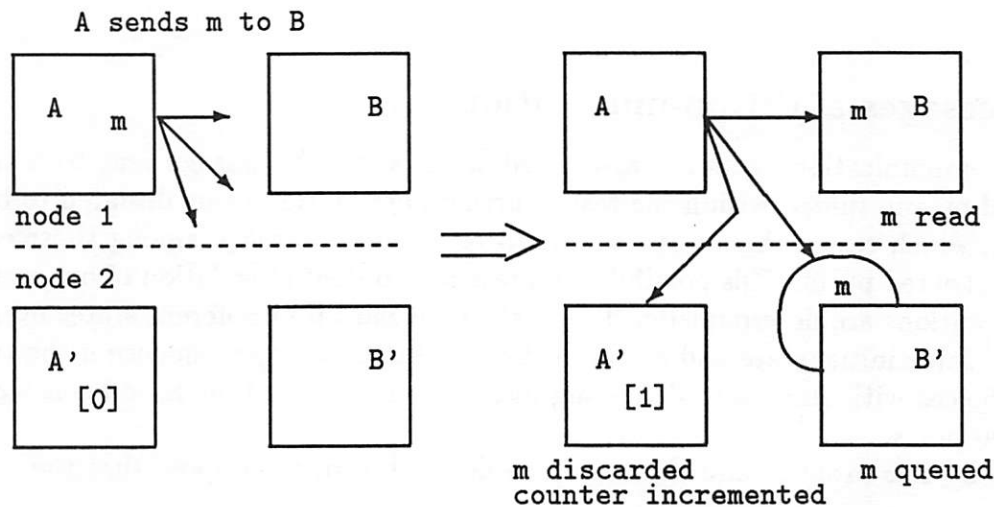


Figure 1: Communication Between Primary and Backup Tasks

enqueued in a manner such that it will be read by the correct thread (yet to be created) if and when the backup has to recover.

The arriving message is treated differently at the three destinations. At the primary destination it is enqueued to be received and processed normally. At the backup destination it is simply enqueued on the corresponding port, if it exists, or on a fictitious port to be created in the future, as explained above. At the backup sender, the message itself is discarded but its arrival is used to increment a counter that represents the number of messages sent since the last checkpoint from the primary to the backup. We call this the *msslc* counter. The kernel primitives

```
msg_send(header, option, timeout)
```

```
msg_rpc(header, option, rcv_size, send_timeout, rcv_timeout)
```

need to be modified such that the *msslc* counter is decremented on each invocation and the specified message not sent as long as the counter value is greater than zero. In this manner, duplicate messages from the primary and its backup are suppressed during recovery.

Since each communication between fault-tolerant tasks involves a 3-way multicast, the overall efficiency of this scheme is critically dependent on the efficiency with which the multicast is realized. In systems such as Auragen and Nixdorf, the protocols necessary for atomic 3-way broadcast are incorporated in the bus arbitration hardware and low-level communication software. Since our system does not necessarily include any such support at the hardware level, it must be realized entirely in software. The obvious place to include an atomic 3-way multicast support in our system is in the network server of Mach. It is this server that extends the Mach IPC abstractions over a network between Mach kernels.



Typically, software solutions to the atomic 3-way multicast problem are complex and expensive in terms of both messages and delay [9]. In its most general formulation, the problem is equivalent to the Byzantine Agreement problem. One property of the hardware we can exploit in achieving efficient atomic multicasts is the existence of a fast broadcast network. The solution we propose is a protocol called *Trans* due to Melliar-Smith and Moser [14].

The basic idea of the *Trans* protocol is the following. Acknowledgements for broadcast messages are placed in messages that are themselves broadcast and are therefore seen by all other nodes. This avoids the necessity of each recipient acknowledging the message separately. In the failure-free case, a broadcast generates two messages—the broadcast message itself and the single (broadcast) acknowledgement.

To understand the protocol, consider the following example where node *A* broadcasts a message<sup>2</sup>.

- Node *B*, among others, receives the message uncorrupted.
- Node *B* includes a positive acknowledgement for *A*'s message in *B*'s next broadcast. If *B* does not have a message to broadcast immediately, it broadcasts just the acknowledgement.
- Node *C* on receiving *B*'s message is aware that *A*'s message has been acknowledged and that there is no need for *C* to acknowledge it again. Node *C*, however, may decide to acknowledge *B*'s message if no other node has acknowledged it yet.
- Suppose a fourth node, *D*, had not received the message broadcast by *A*. The message from *B* alerts *D* of its loss of *A*'s message causing it to include a negative acknowledgement for *A*'s message in its next broadcast.

The protocol generates an extremely small number of acknowledgement messages in practice and has small delays as well. On a high-speed and reliable bus it should deliver satisfactory performance. The protocol can also be extended to guarantee the total ordering requirement of atomic broadcasts as well. The current algorithms for achieving this additional property are computationally intensive and may generate high overhead for the network servers. We feel that this choice of the 3-way multicast protocol needs to be further evaluated and compared with alternatives such as the process group management and broadcast facilities of the ISIS toolkit [4].

## 6.4 Checkpointing

Periodically, the state of the backup task is made to coincide with that of its primary. This checkpointing action is initiated automatically by the kernel of the primary node whenever the primary task has read a certain number of messages. There is also an upper-bound

---

<sup>2</sup>More correctly, the network server task of the node where sender *A* lives broadcasts a message

on the real-time interval that can elapse between checkpoints, regardless of the number of messages read. These two system parameters determine the recovery delay characteristics of the system.

Checkpointing requires the cooperation of the Mach kernels (and perhaps the external pager tasks) at the nodes corresponding to the primary and the backup. The kernel of the primary task needs to keep track of the pages that have been modified since the last checkpoint and send them to the pager of the backup when asked to checkpoint.

The primary kernel is responsible for bringing the execution state of the backup task up-to-date. To accomplish this, the Mach kernel interface needs to be extended to include a

```
checkpoint(header, option, timeout, target_task)
```

primitive to be invoked by one kernel on another. The effect is to establish the execution state of `target_task` managed by the kernel indicated by port `msg_remote_port` (the Mach kernel at the secondary node) to be identical to that specified in the message. The message contains the states (registers, program counter, stack) of all of the threads contained within the task and the number of messages read by the threads in the primary from each of the ports associated with the task since the last checkpoint. The receiving kernel installs the state by creating the ports that do not already exist in the backup, deleting those that have been deallocated, removing the number of messages from the head of each port queue corresponding to the messages already read by the primary and resetting the *msslc* counter to zero.

A prudent policy would be to suspend the primary task until the checkpoint operation is complete. In this manner we have the guarantee that the backup can cleanly take over in case of an eventual failure of the primary. Certain properties of the message delivery system can be exploited to allow checkpointing to go on asynchronously with the primary execution.

## 6.5 Recovery

A failure can affect either a single task (e.g., failure of the CPU executing a thread of the task) or many tasks in a node (e.g., memory module or power failure). In the first case, the Mach kernel at the node where the failed primary was executing sends a `task_resume()` request to the kernel of the backup node to activate the backup. In handling this request, the backup kernel examines the `backup_type` parameter that was specified when the primary-backup task pair was created. If the type is `ANY`, the kernel creates a new backup for the backup (which is about to become the new primary) on any other node that is available. If the type is `HERE`, a new backup will be created on the same node as the failed primary (perhaps after waiting for the node to come up if it had failed completely). This may be dictated by tasks that are associated with the physical resources particular to that node and can only be restarted there<sup>3</sup>. Finally, if the type is `ONCE` no new backup is created once a primary fails.

<sup>3</sup>In [6], the authors note that since certain system server tasks have to be type `HERE`, it makes little sense to provide type `ANY` as the overall system availability cannot exceed those of the server tasks

In all cases, the kernel completes the resume operation by making the threads of the task executable. The network servers of the entire system are notified of the recovery so as to be able to construct the new members of the 3-way multicasts involving this task. The backup recovers simply by executing. Given that it starts from a checkpoint state and reads the same messages that the primary received, it eventually reaches the same state as the primary before its failure. Recall that the *msslc* counter prevents the recovering backup to resend messages already sent by the primary.

In case of a node failure involving multiple tasks, the entire node is declared down and each Mach kernel examines if it contains any tasks that are backups for primaries on the failed node. For each such task the above recovery procedure is performed.

Failure detection and notification is carried out by a collaboration between the nodes using a periodic polling mechanism.

## 7 Discussion of Remaining Problems

In addition to the non-determinism of the message system discussed in Section 6.3, the fact that Mach tasks can be multi-threaded with shared memory can also lead to non-deterministic executions. When a task has multiple runnable threads, the scheduling among them can be arbitrary. Consequently, two initially identical tasks may have very different final states depending on the order in which their threads executed. The Mach scheduler must be modified to make thread scheduling deterministic. In a manner analogous to our solution for the message receive non-determinism, we propose that the scheduler discriminate between otherwise “equal” threads by picking the one with the smaller port identifier. Note that the scheduler can use whatever other criterion (e.g., priority) to partially order threads—the above rule is invoked only at the very end to break remaining ties.

Typical operating system services such as `time_of_day()` and `get_pid()` can be sources for non-deterministic computations. In Mach, all global services including time-of-day are handled by server tasks through the usual message communication mechanism. Consequently, a request for the time-of-day is guaranteed to return the same value to both the primary and the backup through the mechanisms outlined above. In Mach, all references to objects are done indirectly through (protected) ports. Thus, the equivalent of the “pid” in Mach is task port which can only be used indirectly in referencing and its value cannot be examined.

Finally, interactions between Mach tasks and the external world can lead to non-deterministic computations. These interactions can be explicit as in the case of signals being generated by a user from the keyboard, or implicit as in the case of page faults resulting from memory contention. Coping with these sources of non-determinism as well as evaluating the consequences of rendering scheduling deterministic remain open questions. In the worst case, we will have to confront non-deterministic computations squarely. The costs associated with this alternative are high. One possibility, as advocated by the Clouds system [10], is to actively replicate computations (thus consume processing power even in the absence of failures) and invoke an algorithm to elect one of the (correct but possibly different) outputs to

commit as "the" output of the computation. Alternatively, we could adopt the algorithms of [12] and pay the price as complex checkpointing and rollback operations. Not only are the primitives more complex, rollback involves more computations since we can no longer rely on "replaying" recorded messages to generate the same sequence of messages.

**Acknowledgements** This work benefited from many discussions the author had with Giuseppe Malatesta and Giampiero Paolillo of the Olivetti Research Laboratory, Trezzano, Milano.

## References

- [1] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of Summer Usenix*, pp. 93-112, July 1986.
- [2] J. F. Bartlett. A NonStop Kernel. *Proc. Eighth Symposium on Operating Systems Principles*, pp. 22-29, Asilomar, California, December 1981.
- [3] P. A. Bernstein. Sequioa: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing. *IEEE Computer*, 21(2), pp. 37-45, February 1988.
- [4] K. P. Birman. Replication and Fault-Tolerance in the ISIS System. In *Proc. Tenth Symposium on Operating System Principles*, pp. 79-86, Orcas Island, Washington, December 1985.
- [5] A. Borg, J. Baumbach and S. Glazer. A Message System for Supporting Fault Tolerance. In *Proc. Ninth Symposium on Operating System Principles*, pp. 90-99, Bretton Woods, N. H., October 1983.
- [6] A. Borg, W. Blau, W. Graetsch, F. Herrmann and W. Oberle. Fault Tolerance Under UNIX. *ACM Transaction on Computer Systems*, 7(1), pp. 1-24, February 1989.
- [7] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States in a Distributed System. *ACM Transaction on Computer Systems*, 3(1), pp. 63-75, February 1985.
- [8] D. R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3), pp. 314-333, March 1988.
- [9] F. Cristian, H. Aghili and R. Strong. Atomic Broadcasts: From Simple Message Diffusion to Byzantine Agreement. In *Proc. 15th International Symposium on Fault-Tolerant Computing*, pp. 200-206, July 1985.

- [10] P. Dasgupta, R. J. LeBlanc and E. Spafford. The Clouds Project: Design and Implementation of a Fault-Tolerant Distributed Operating System. Technical Report, Georgia Institute of Technology, Atlanta, 1985.
- [11] J. Gray, P. McJones, M. Blasgen, B. Linsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2), pp. 223–242, June 1981.
- [12] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transaction on Software Engineering*, SE-13(1), pp. 23–31, January 1987.
- [13] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler and W. Weihl. Argus Reference Manual. Technical Report MIT/LCS/TR-400, Massachusetts Institute of Technology, Cambridge, Massachusetts, November 1987.
- [14] P. M. Melliar-Smith and L. E. Moser. Trans: A Broadcast Protocol for Distributed Systems. Technical Report TRCS88-28, Department of Computer Science, University of California, Santa Barbara, December 1988.
- [15] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing* MIT Press, Boston, Massachusetts, 1985.
- [16] S. J. Mullender and A. S. Tanenbaum. The Design of a Capability-Based Distributed Operating System. *The Computer Journal*, 29(4), pp. 289–300, 1986.
- [17] M. Rozier, *et al.* Chorus Distributed Operating System. In *Computing Systems*, 1(4), 1988.
- [18] S. K. Shrivastava, G. N. Dixon, G. D. Parrington, F. Hedayati, S. M. Wheeler and M. C. Little. The Design and Implementation of Arjuna. Technical Report, Computing Laboratory, University of Newcastle upon Tyne, March 1989.
- [19] A. Z. Spector, D. S. Daniels, D. J. Duchamp, J. L. Eppinger and R. Pausch. Distributed Transactions for Reliable Systems. In *Proc. Tenth Symposium on Operating System Principles*, pp. 127–146, Orcas Island, Washington, December 1985.
- [20] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transaction on Computer Systems*, 3(3), pp. 204–226, August 1985.
- [21] D. Taylor and G. Wilson. Stratus. In *Dependability of Resilient Computers*, pp. 222–236, T. Anderson (Ed.), BSP Professional Books, Oxford, England, 1989.
- [22] Tolerant Systems, Inc. Eternity Series: Technology Brief. Internal publication, Tolerant Systems, San Jose, California, July 1988.
- [23] D. B. Johnson and W. Zwaenepoel. Sender-Based Message Logging. In *Proc. 17th International Symposium on Fault-Tolerant Computing*, pp. 14–19, Pittsburgh, Pennsylvania, July 1987.





## *The USENIX Association*

The USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *login*; produces a quarterly technical journal, *Computing Systems*; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts. Most recently, the Association created UUNET Communications Services, Inc., a separate not-for-profit organization offering electronic communications services to those wishing to participate in the UNIX milieu.

*Computing Systems*, published quarterly by the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX technical community. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association, differentiated primarily by the dues paid and services provided.

For further information about membership or to order publications, contact:

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710

Telephone: 415 528-8649  
Email: [office@usenix.org](mailto:office@usenix.org)

### *USENIX Supporting Members*

Aerospace Corporation  
AT&T Information Systems  
Digital Equipment Corporation  
Frame Technology Corporation

mt Xinu  
Open Software Foundation  
Quality Micro Systems  
Sun Microsystems, Inc.  
Sybase, Inc.

